MongoDB Documentation

Release 2.2.1

MongoDB Documentation Project

November 12, 2012

CONTENTS

Part I

About MongoDB Documentation

CHAPTER

ONE

ABOUT MONGODB

MongoDB is a *document*-oriented database management system designed for performance, horizontal scalability, high availability, and advanced queryability. See the following wiki pages for more information about MongoDB:

- Introduction
- Philosophy
- About

If you want to download MongoDB, see the downloads page.

If you'd like to learn how to use MongoDB with your programming language of choice, see the introduction to the *drivers* (page 285).

CHAPTER

TWO

ABOUT THE DOCUMENTATION PROJECT

2.1 This Manual

The MongoDB documentation project provides a complete manual for the MongoDB database. This resource is replacing eventually replace MongoDB's original documentation.

2.1.1 Licensing

This manual is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported" (i.e. "CC-BY-NC-SA") license.

2.1.2 Version and Revisions

This version of the manual reflects version 2.2.1 of MongoDB.

See the MongoDB Documentation Project Page 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.

This edition reflects "`master`" branch of the documentation as of the "`759748029411a63acc689e398c8dfe6b65d9b133`" revision. This branch is explicitly accessible via "http://docs.mongodb.org/master" and you can always reference the commit of the current manual in the release.txt file.

The most up-to-date, current, and stable version of the manual is always available at "http://docs.mongodb.org/manual/."

2.2 Contributing to the Documentation

The entire source of the documentation is available in the docs repository along with all of the other MongoDB project repositories on GitHub. You can clone the repository by issuing the following command at your system shell:

```
git clone git://github.com/mongodb/docs.git
```

If you have a GitHub account and want to fork this repository, you may issue pull requests, and someone on the documentation team will merge in your contributions promptly. In order to accept your changes to the Manual, you have to complete the MongoDB/10gen Contributor Agreement.

This project tracks issues at MongoDB's DOCS project. If you see a problem with the documentation, please report it there.

2.3 Writing Documentation

The MongoDB Manual uses Sphinx, a sophisticated documentation engine built upon Python Docutils. The original reStructured Text files, as well as all necessary Sphinx extensions and build tools, are available in the same repository as the documentation.

You can view the documentation style guide and the build instructions in reStructured Text files in the top-level of the documentation repository. If you have any questions, please feel free to open a Jira Case.

Part II

Installing MongoDB

CHAPTER

THREE

INSTALLATION GUIDES

MongoDB runs on most platforms, and supports 32-bit and 64-bit architectures. 10gen, the MongoDB makers, provides both binaries and packages. Choose your platform below:

3.1 Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux

3.1.1 Synopsis

This tutorial outlines the basic installation process for deploying *MongoDB* on Red Hat Enterprise Linux, CentOS Linux, Fedora Linux and related systems. This procedure uses `.rpm` packages as the basis of the installation. 10gen publishes packages of the MongoDB releases as `.rpm` packages for easy installation and management for users of Debian systems. While some of these distributions include their own MongoDB packages, the 10gen packages are generally more up to date.

This tutorial includes: an overview of the available packages, instructions for configuring the package manager, the process install packages from the 10gen repository, and preliminary MongoDB configuration and operation.

See Also:

The documentation of following related processes and concepts.

Other installation tutorials:

- `http://docs.mongodb.org/manual/tutorial/install-mongodb-on-debian-or-ubuntu-linux`
- *Install MongoDB on Debian* (page 15)
- *Install MongoDB on Ubuntu* (page 12)
- *Install MongoDB on Linux* (page 17)
- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 22)

3.1.2 Package Options

The 10gen repository contains four packages:

- `mongo-10gen`

  This package contains MongoDB tools from latest **stable** release. Install this package on all production MongoDB hosts and optionally on other systems from which you may need to administer MongoDB systems.

- `mongo-10gen-server`

  This package contains the `mongod` and `mongos` (page 676) daemons from the latest **stable** release and associated configuration and init scripts.

- `mongo18-10gen`

  This package contains MongoDB tools from previous release. Install this package on all production MongoDB hosts and optionally on other systems from which you may need to administer MongoDB systems.

- `mongo18-10gen-server`

  This package contains the `mongod` and `mongos` (page 676) daemons from previous stable release and associated configuration and init scripts.

The MongoDB tools included in the `mongo-10gen` packages are:

- `mongo`
- `mongodump`
- `mongorestore`
- `mongoexport`
- `mongoimport`
- `mongostat`
- `mongotop`
- `bsondump`

3.1.3 Installing MongoDB

Configure Package Management System (YUM)

Create a `http://docs.mongodb.org/manual/etc/yum.repos.d/10gen.repo` file to hold information about your repository. If you are running a 64-bit system (recommended,) place the following configuration in `http://docs.mongodb.org/manual/etc/yum.repos.d/10gen.repo` file:

```
[10gen]
name=10gen 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 isn't recommended for production deployments, place the following configuration in `http://docs.mongodb.org/manual/etc/yum.repos.d/10gen.repo` file:

```
[10gen]
name=10gen Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/i686
gpgcheck=0
enabled=1
```

Installing 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! Continue for configuration and start-up suggestions.

3.1.4 Configure MongoDB

These packages configure MongoDB using the `http://docs.mongodb.org/manual/etc/mongod.conf` file in conjunction with the *control script*. You can find the init script at `http://docs.mongodb.org/manual/etc/rc.d/init.d/mongod`.

This MongoDB instance will store its data files in the `http://docs.mongodb.org/manual/var/lib/mongo` and its log files in `http://docs.mongodb.org/manual/var/log/mongo`, and run using the `mongod` user account.

Note: If you change the user that runs the MongoDB process, you will need to modify the access control rights to the `http://docs.mongodb.org/manual/var/lib/mongo` and `http://docs.mongodb.org/manual/var/log/mongo` directories.

3.1.5 Control MongoDB

Start MongoDB

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 `http://docs.mongodb.org/manual/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 `http://docs.mongodb.org/manual/var/log/mongo/mongod` file to watch for errors or important messages from the server.

Control mongos

As of the current release, there are no *control scripts* for mongos (page 676). mongos (page 676) is only used in sharding deployments and typically do not run on the same systems where mongod runs. You can use the mongodb script referenced above to derive your own mongos (page 676) control script.

3.1.6 Using MongoDB

Among the tools included in the mongo-10gen package, is the mongo shell. You can connect to your MongoDB instance by issuing the following command at the system prompt:

```
mongo
```

This will connect to the database running on the localhost interface by default. At the mongo prompt, issue the following two commands to insert a record in the "test" *collection* of the (default) "test" database and then retrieve that document.

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

See Also:

"mongo" and "http://docs.mongodb.org/manual/reference/javascript"

3.2 Install MongoDB on Ubuntu

3.2.1 Synopsis

This tutorial outlines the basic installation process for installing *MongoDB* on Ubuntu Linux systems. This tutorial uses .deb packages as the basis of the installation. 10gen publishes packages of the MongoDB releases as .deb packages for easy installation and management for users of Ubuntu systems. Ubuntu does include MongoDB packages, the 10gen packages are generally more up to date.

This tutorial includes: an overview of the available packages, instructions for configuring the package manager, the process for installing packages from the 10gen repository, and preliminary MongoDB configuration and operation.

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 15) tutorial.

See Also:

The documentation of following related processes and concepts.

Other installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 9)
- *Install MongoDB on Debian* (page 15)
- *Install MongoDB on Linux* (page 17)
- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 22)

3.2.2 Package Options

The 10gen repository contains three packages:

- `mongodb-10gen`

  This package contains the latest **stable** release. Use this for production deployments.

- `mongodb20-10gen`

  This package contains the stable release of v2.0 branch.

- `mongodb18-10gen`

  This package contains the stable release of v1.8 branch.

You cannot install these packages concurrently with each other or with the `mongodb` package that your release of Ubuntu may include.

10gen also provides packages for "unstable" or development versions of MongoDB. Use the `mongodb-10gen-unstable` package to test the latest development release of MongoDB, but do not use this version in production.

3.2.3 Installing 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 10gen public GPG Key:

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

Create a `http://docs.mongodb.org/manual/etc/apt/sources.list.d/10gen.list` file and include the following line for the 10gen repository.

```
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen
```

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.

3.2.4 Configure MongoDB

These packages configure MongoDB using the `http://docs.mongodb.org/manual/etc/mongodb.conf` file in conjunction with the *control script*. You will find the control script is at `http://docs.mongodb.org/manual/etc/init.d/mongodb`.

This MongoDB instance will store its data files in the http://docs.mongodb.org/manual/var/lib/mongodb and its log files in http://docs.mongodb.org/manual/var/log/mongodb, and run using the mongodb user account.

Note: If you change the user that runs the MongoDB process, you will need to modify the access control rights to the http://docs.mongodb.org/manual/var/lib/mongodb and http://docs.mongodb.org/manual/var/log/mongodb directories.

3.2.5 Controlling MongoDB

Starting 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 http://docs.mongodb.org/manual/var/log/mongodb/mongodb.log.

Stopping MongoDB

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

```
sudo service mongodb stop
```

Restarting MongoDB

You may restart the mongod process by issuing the following command:

```
sudo service mongodb restart
```

Controlling mongos

As of the current release, there are no *control scripts* for mongos (page 676). mongos (page 676) is only used in sharding deployments and typically do not run on the same systems where mongod runs. You can use the mongodb script referenced above to derive your own mongos (page 676) control script.

3.2.6 Using MongoDB

Among the tools included with the MongoDB package, is the mongo shell. You can connect to your MongoDB instance by issuing the following command at the system prompt:

```
mongo
```

This will connect to the database running on the localhost interface by default. At the mongo prompt, issue the following two commands to insert a record in the "test" *collection* of the (default) "test" database.

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

See Also:

"mongo" and "http://docs.mongodb.org/manual/reference/javascript"

3.3 Install MongoDB on Debian

3.3.1 Synopsis

This tutorial outlines the basic installation process for installing *MongoDB* on Debian systems. This tutorial uses .deb packages as the basis of the installation. 10gen publishes packages of the MongoDB releases as .deb packages for easy installation and management for users of Debian systems. While some of these distributions include their own MongoDB packages, the 10gen packages are generally more up to date.

This tutorial includes: an overview of the available packages, instructions for configuring the package manager, the process for installing packages from the 10gen repository, and preliminary MongoDB configuration and operation.

Note: If you're running a version of Ubuntu Linux prior to 9.10 "Karmic," use this tutorial. Other Ubuntu users will want to follow the *Install MongoDB on Ubuntu* (page 12) tutorial.

See Also:

The documentation of following related processes and concepts.

Other installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 9)
- *Install MongoDB on Ubuntu* (page 12)
- *Install MongoDB on Linux* (page 17)
- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 22)

3.3.2 Package Options

The 10gen repository contains three packages:

- mongodb-10gen

  This package contains the latest **stable** release. Use this for production deployments.

- mongodb20-10gen

  This package contains the stable release of v2.0 branch.

- mongodb18-10gen

  This package contains the stable release of v1.8 branch.

You cannot install these packages concurrently with each other or with the mongodb package that your release of Debian may include.

10gen also provides packages for "unstable" or development versions of MongoDB. Use the mongodb-10gen-unstable package to test the latest development release of MongoDB, but do not use this version in production.

3.3.3 Installing MongoDB

Configure Package Management System (APT)

The Debian 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 10gen public GPG Key:

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

Create a the `http://docs.mongodb.org/manual/etc/apt/sources.list.d/10gen.list` file and include the following line for the 10gen repository.

```
deb http://downloads-distro.mongodb.org/repo/debian-sysvinit dist 10gen
```

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.

3.3.4 Configure MongoDB

These packages configure MongoDB using the `http://docs.mongodb.org/manual/etc/mongodb.conf` file in conjunction with the *control script*. You can find the control script at `http://docs.mongodb.org/manual/etc/init.d/mongodb`.

This MongoDB instance will store its data files in the `http://docs.mongodb.org/manual/var/lib/mongodb` and its log files in `http://docs.mongodb.org/manual/var/log/mongodb`, and run using the `mongodb` user account.

Note: If you change the user that runs the MongoDB process, you will need to modify the access control rights to the `http://docs.mongodb.org/manual/var/lib/mongodb` and `http://docs.mongodb.org/manual/var/log/mongodb` directories.

3.3.5 Controlling MongoDB

Starting 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 `http://docs.mongodb.org/manual/var/log/mongodb/mongodb.log`.

Stopping MongoDB

Issue the following command to stop `mongod`:

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

Restarting MongoDB

Issue the following command to restart `mongod`:

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

Controlling `mongos`

As of the current release, there are no *control scripts* for `mongos` (page 676). `mongos` (page 676) is only used in sharding deployments and typically do not run on the same systems where `mongod` runs. You can use the `mongodb` script referenced above to derive your own `mongos` (page 676) control script.

3.3.6 Using MongoDB

Among the tools included with the MongoDB package, is the `mongo` shell. You can connect to your MongoDB instance by issuing the following command at the system prompt:

```
mongo
```

This will connect to the database running on the localhost interface by default. At the `mongo` prompt, issue the following two commands to insert a record in the "test" *collection* of the (default) "test" database.

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

See Also:

"`mongo`" and "`http://docs.mongodb.org/manual/reference/javascript`"

3.4 Install MongoDB on Linux

3.4.1 Synopsis

10gen provides compiled versions of *MongoDB* for use on Linux that provides a simple option for users who cannot use packages. This tutorial outlines the basic installation of MongoDB using these compiled versions and an initial usage guide.

See Also:

The documentation of following related processes and concepts.

Other installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 9)
- *Install MongoDB on Ubuntu* (page 12)
- *Install MongoDB on Debian* (page 15)

- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 22)

3.4.2 Download MongoDB

Note: You should place the MongoDB binaries in a central location on the file system that is easy to access and control. Consider `http://docs.mongodb.org/manual/opt` or `http://docs.mongodb.org/manual/usr/local/bin`.

In a terminal session, begin by downloading the latest release. In most cases you will want to download the 64-bit version of MongoDB.

```
curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.2.1.tgz > mongo.tgz
```

If you need to run the 32-bit version, use the following command.

```
curl http://downloads.mongodb.org/linux/mongodb-linux-i686-2.2.1.tgz > mongo.tgz
```

Once you've downloaded the release, issue the following command to extract the files from the archive:

```
tar -zxvf mongo.tgz
```

Optional

You may use the following command to copy the extracted folder into a more generic location.

```
cp -R -n  mongodb-osx-20??-??-??/ mongodb
```

You can find the `mongod` binary, and the binaries all of the associated MongoDB utilities, in the `bin/` directory within the extracted directory.

Using MongoDB

Before you start `mongod` for the first time, you will need to create the data directory. By default, `mongod` writes data to the `http://docs.mongodb.org/manual/data/db/` directory. To create this directory, use the following command:

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

Note: Ensure that the system account that will run the `mongod` process has read and write permissions to this directory. If `mongod` runs under the `mongo` user account, issue the following command to change the owner of this folder:

```
chown mongo /data/db
```

If you use an alternate location for your data directory, ensure that this user can write to your chosen data path.

You can specify, and create, an alternate path using the *--dbpath* (page 583) option to `mongod` and the above command.

The 10gen builds of MongoDB contain no *control scripts* or method to control the `mongod` process. You may wish to create control scripts, modify your path, and/or create symbolic links to

the MongoDB programs in your `http://docs.mongodb.org/manual/usr/local/bin` or `http://docs.mongodb.org/manual/usr/bin` directory for easier use.

For testing purposes, you can start a `mongod` directly in the terminal without creating a control script:

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

Note: The above command assumes that the `mongod` binary is accessible via your system's search path, and that you have created a default configuration file located at `http://docs.mongodb.org/manual/etc/mongod.conf`.

Among the tools included with this MongoDB distribution, is the `mongo` shell. You can use this shell to connect to your MongoDB instance by issuing the following command at the system prompt:

```
./bin/mongo
```

Note: The `./bin/mongo` command assumes that the `mongo` binary is in the `bin/` sub-directory of the current directory. This is the directory into which you extracted the `.tgz` file.

This will connect to the database running on the localhost interface by default. At the `mongo` 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/javascript`"

3.5 Install MongoDB on OS X

3.5.1 Synopsis

This tutorial outlines the basic installation process for deploying *MongoDB* on Macintosh OS X systems. This tutorial provides two main methods of installing the MongoDB server (i.e. "`mongod`") and associated tools: first using the community package management tools, and second using builds of MongoDB provided by 10gen.

See Also:

The documentation of following related processes and concepts.

Other installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 9)
- *Install MongoDB on Ubuntu* (page 12)
- *Install MongoDB on Debian* (page 15)
- *Install MongoDB on Linux* (page 17)
- *Install MongoDB on Windows* (page 22)

3.5.2 Installing with Package Management

Both community package management tools: Homebrew and MacPorts require some initial setup and configuration. This configuration is beyond the scope of this document. You only need to use one of these tools.

If you want to use package management, and do not already have a system installed, Homebrew is typically easier and simpler to use.

Homebrew

Homebrew installs binary packages based on published "formula." Issue the following command at the system shell to update the `brew` package manager:

```
brew update
```

Use the following command to install the MongoDB package into your Homebrew system.

```
brew install mongodb
```

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

```
brew update
brew upgrade mongodb
```

MacPorts

MacPorts distributes build scripts that allow you to easily build packages and their dependencies on your own system. The compilation process can take significant period of time depending on your system's capabilities and existing dependencies. Issue the following command in the system shell:

```
port install mongodb
```

Using MongoDB from Homebrew and MacPorts

The packages installed with Homebrew and MacPorts contain no *control scripts* or interaction with the system's process manager.

If you have configured Homebrew and MacPorts correctly, including setting your `PATH`, the MongoDB applications and utilities will be accessible from the system shell. Start the `mongod` process in a terminal (for testing or development) or using a process management tool.

```
mongod
```

Then open the `mongo` shell by issuing the following command at the system prompt:

```
mongo
```

This will connect to the database running on the localhost interface by default. At the `mongo` 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/javascript"

3.5.3 Installing from 10gen Builds

10gen provides compiled binaries of all MongoDB software compiled for OS X, which may provide a more straightforward installation process.

Download MongoDB

In a terminal session, begin by downloading the latest release. Use the following command at the system prompt:

```
curl http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.2.1.tgz > mongo.tgz
```

Note: The mongod process will not run on older Macintosh computers with PowerPC (i.e. non-Intel) processors.

Once you've downloaded the release, issue the following command to extract the files from the archive:

```
tar -zxvf mongo.tgz
```

Optional

You may use the following command to move the extracted folder into a more generic location.

```
mv -n mongodb-osx-[platform]-[version]/ /path/to/new/location/
```

Replace [platform] with i386 or x86_64 depending on your system and the version you downloaded, and [version] with 2.2.1 or the version of MongoDB that you are installing.

You can find the mongod binary, and the binaries all of the associated MongoDB utilities, in the bin/ directory within the archive.

Using MongoDB from 10gen Builds

Before you start mongod for the first time, you will need to create the data directory. By default, mongod writes data to the http://docs.mongodb.org/manual/data/db/ directory. To create this directory, and set the appropriate permissions use the following commands:

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

You can specify an alternate path for data files using the *--dbpath* (page 583) option to mongod.

The 10gen builds of MongoDB contain no *control scripts* or method to control the mongod process. You may wish to create control scripts, modify your path, and/or create symbolic links to the MongoDB programs in your http://docs.mongodb.org/manual/usr/local/bin directory for easier use.

For testing purposes, you can start a mongod directly in the terminal without creating a control script:

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

Note: This command assumes that the `mongod` binary is accessible via your system's search path, and that you have created a default configuration file located at `http://docs.mongodb.org/manual/etc/mongod.conf`.

Among the tools included with this MongoDB distribution, is the `mongo` shell. You can use this shell to connect to your MongoDB instance by issuing the following command at the system prompt from inside of the directory where you extracted `mongo`:

```
./bin/mongo
```

Note: The `./bin/mongo` command assumes that the `mongo` binary is in the `bin/` sub-directory of the current directory. This is the directory into which you extracted the `.tgz` file.

This will connect to the database running on the localhost interface by default. At the `mongo` 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/javascript`"

3.6 Install MongoDB on Windows

3.6.1 Synopsis

This tutorial provides a method for installing and running the MongoDB server (i.e. "`mongod.exe`") on the Microsoft Windows platform through the *Command Prompt* and outlines the process for setting up MongoDB as a *Windows Service*.

Operating MongoDB with Windows is similar to MongoDB on other platforms. Most components share the same operational patterns.

3.6.2 Procedure

Download MongoDB for Windows

Download the latest production release of MongoDB from the MongoDB downloads page.

There are three builds of MongoDB for Windows:

- MongoDB for Windows Server 2008 R2 edition 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.

Changed 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.

Note: Always download the correct version of MongoDB for your Windows system. The 64-bit versions of MongoDB will not work with 32-bit Windows.

32-bit versions of MongoDB are suitable only for testing and evaluation purposes and only support databases smaller than 2GB.

You can find the architecture of your version of Windows platform using the following command in the *Command Prompt*

```
wmic os get osarchitecture
```

In Windows Explorer, find the MongoDB download file, typically in the default Downloads directory. Extract the archive to `C:\` by right clicking on the archive and selecting *Extract All* and browsing to `C:\`.

Note: The folder name will be either:

```
C:\mongodb-win32-i386-[version]
```

Or:

```
C:\mongodb-win32-x86_64-[version]
```

In both examples, replace `[version]` with the version of MongoDB downloaded.

Set up the Environment

Start the *Command Prompt* by selecting the *Start Menu*, then *All Programs*, then *Accessories*, then right click *Command Prompt*, and select *Run as Administrator* from the popup menu. 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`)

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*. Issue the following command sequence:

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

Note: You may specify an alternate path for `\data\db` with the `dbpath` (page 624) 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"
```

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 and Authentication wiki page.

Warning: Do not allow `mongod.exe` to be accessible to public networks without running in "Secure Mode" (i.e. `auth` (page 624).) MongoDB is designed to be run in "trusted environments" and the database does not enable authentication or "Secure Mode" by default.

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/javascript`." If you want to develop applications using .NET, see the C# Language Center wiki page for more information.

3.6.3 MongoDB as a Windows Service

New in version 2.0. Setup 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

You should specify two options when running MongoDB as a Windows Service: a path for the log output (i.e. `logpath` (page 622)) and a *configuration file* (page 621).

1. Create a specific directory for MongoDB log files:

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

2. Create a configuration file for the `logpath` (page 622) option for MongoDB in the *Command Prompt* by issuing this command:

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

While these optional steps are optional, creating a specific location for log files and using the configuration file are good practice.

Note: Consider setting the `logappend` (page 623) 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` (page 622) and `logappend` (page 623) behavior will change in the 2.2 release.

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`* (page 594) option to succeed, you *must* specify a `logpath` (page 622) setting or the *`--logpath`* (page 582) run-time option.

2. To run the MongoDB service:

   ```
   net start MongoDB
   ```

Note: If you wish to use an alternate path for your `dbpath` (page 624) specify it in the config file (e.g. `C:\mongodb\mongod.cfg`) on that you specified in the *`--install`* (page 594) operation. You may also specify *`--dbpath`* (page 583) on the command line; however, always prefer the configuration file.

If the `dbpath`' directory does not exist, `mongod.exe` will not be able to start. The default value for `dbpath` (page 624) 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
  ```

CHAPTER

FOUR

RELEASE NOTES

You should always install the latest, stable version of MongoDB. Stable versions have an even-numbered minor version number. For example: v2.2 is stable, v2.0 and v1.8 were previously the stable, while v2.1 and v2.3 is a development version.

- Current Stable Release
  - *Release Notes for MongoDB 2.2* (page 693)
- Previous Stable Releases
  - *Release Notes for MongoDB 2.0* (page 703)
  - *Release Notes for MongoDB 1.8* (page 709)
- Current Development Release (v2.3-series)

Part III

Replication

Database replication ensures redundancy, backup, and automatic failover. Replication occurs through groups of servers known as replica sets.

This page lists the documents, tutorials, and reference pages that describe replica sets.

For an overview, see *Replication Fundamentals* (page 33). To work with members, see *Replica Set Administration* (page 38). To configure deployment architecture, see *Replication Architectures* (page 51). To modify read and write operations, see *Application Development with Replica Sets* (page 54). For procedures for performing certain replication tasks, see the *list of replication tutorials* (page 73).

CHAPTER

FIVE

DOCUMENTATION

The following is the outline of the main documentation:

5.1 Replication Fundamentals

A MongoDB *replica set* is a cluster of `mongod` instances that replicate amongst one another and ensure automated failover. Most replica sets consists of two or more `mongod` instances with at most one of these designated as the primary and the rest as secondary members. Clients direct all writes to the primary, while the secondary members replicate from the primary asynchronously.

Database replication with MongoDB adds redundancy, helps to ensure high availability, simplifies certain administrative tasks such as backups, and may increase read capacity. Most production deployments use replication.

If you're familiar with other database systems, you may think about replica sets as a more sophisticated form of traditional master-slave replication. [1] In master-slave replication, a *master* node accepts writes while one or more *slave* nodes replicate those write operations and thus maintain data sets identical to the master. For MongoDB deployments, the member that accepts write operations is the **primary**, and the replicating members are **secondaries**.

MongoDB's replica sets provide automated failover. If a *primary* fails, the remaining members will automatically try to elect a new primary.

A replica set can have up to 12 members, but only 7 members can have votes. For information regarding non-voting members, see *non-voting members* (page 42)

See Also:

The *Replication* (page 31) index for a list of the documents in this manual that describe the operation and use of replica sets.

5.1.1 Member Configuration Properties

You can configure replica set members in a variety of ways, as listed here. In most cases, members of a replica set have the default proprieties.

- **Secondary-Only**: These members have data but cannot become primary under any circumstance. See *Secondary-Only Members* (page 39).
- **Hidden**: These members are invisible to client applications. See *Hidden Members* (page 40).

[1] MongoDB also provides conventional master/slave replication. Master/slave replication operates by way of the same mechanism as replica sets, but lacks the automatic failover capabilities. 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 11 *slave* members, you'll need to use master/slave replication.

- **Delayed**: These members apply operations from the primary's *oplog* after a specified delay. You can think of a delayed member as a form of "rolling backup." See *Delayed Members* (page 40).
- **Arbiters**: These members have no data and exist solely to participate in *elections* (page 34). See *Arbiters* (page 41).
- **Non-Voting**: These members do not vote in elections. Non-voting members are only used for larger sets with more than 12 members. See *Non-Voting Members* (page 42).

For more information about each member configuration, see the *Member Configurations* (page 39) section in the *Replica Set Administration* (page 38) document.

5.1.2 Failover

Replica sets feature automated failover. If the *primary* goes offline or becomes unresponsive and a majority of the original set members can still connect to each other, the set will elect a new primary.

For a detailed explanation of failover, see the *Failover and Recovery* (page 49) section in the *Replica Set Administration* (page 38) document.

Elections

When any failover occurs, an election takes place to decide which member should become primary.

Elections provide a mechanism for the members of a *replica set* to autonomously select a new *primary* without administrator intervention. The election allows replica sets to recover from failover situations very quickly and robustly.

Whenever the primary becomes unreachable, the secondary members trigger an election. The first member to receive votes from a majority of the set will become primary. The most important feature of replica set elections is that a majority of the original number of members in the replica set must be present for election to succeed. If you have a three-member replica set, the set can elect a primary when two or three members can connect to each other. If two members in the replica go offline, then the remaining member will remain a secondary.

Note: When the current *primary* steps down and triggers an election, the `mongod` instances will close all client connections. This ensures that the clients maintain an accurate view of the *replica set* and helps prevent *rollbacks*.

For more information on elections and failover, see:

- The *Failover and Recovery* (page 49) section in the *Replica Set Administration* (page 38) document.
- The *Election Internals* (page 63) section in the *Replication Internals* (page 61) document

Member Priority

In a replica set, every member has a "priority," that helps determine eligibility for *election* (page 34) to *primary*. By default, all members have a priority of `1`, unless you modify the `members[n].priority` (page 663) value. All members have a single vote in elections.

Warning: Always configure the `members[n].priority` (page 663) value to control which members will become primary. Do not configure `members[n].votes` (page 664) except to permit more than 7 secondary members.

For more information on member priorities, see the *Adjusting Priority* (page 43) section in the *Replica Set Administration* (page 38) document.

5.1.3 Consistency

This section provides an overview of the concepts that underpin database consistency and the MongoDB mechanisms to ensure that users have access to consistent data.

In MongoDB, all read operations issued to the primary of a replica set are *consistent* with the last write operation.

If clients configure the *read preference* to permit allow secondary reads, read operations cannot return from *secondary* members that have not replicated more recent updates or operations. In these situations the query results may reflect 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.

There is no way to guarantee consistency for reads from *secondary members,* except by configuring the *client* and *driver* to ensure that write operations succeed on all members before completing successfully.

Rollbacks

In some *failover* situations *primaries* will have accepted write operations that have *not* replicated to the *secondaries* after a failover occurs. This case is rare and typically occurs as a result of a network partition with replication lag. When this member (the former primary) rejoins the *replica set* and attempts to continue replication as a secondary the former primary must revert these operations or "roll back" these operations to maintain database consistency across the replica set.

MongoDB writes the rollback data to a *BSON* file in the database's `dbpath` (page 624) directory. Use *bsondump* (page 602) to read the contents of these rollback files and then manually apply the changes to the new primary. There is no way for MongoDB to appropriately and fairly handle rollback situations without manual intervention. Even after the member completes the rollback and returns to secondary status, administrators will need to apply or decide to ignore the rollback data.

The best strategy for avoiding all rollbacks is to ensure *write propagation* (page 54) to all or some of the members in the set. Using these kinds of policies prevents situations that might create rollbacks.

Warning: A `mongod` instance will not rollback more than 300 megabytes of data. If your system needs to rollback more than 300 MB, you will need to manually intervene to recover this data.

For more information on failover, see:

- The *Failover* (page 34) section in this document.
- The *Failover and Recovery* (page 49) section in the *Replica Set Administration* (page 38) document.

Application Concerns

Client applications are indifferent to the configuration and operation of replica sets. While specific configuration depends to some extent on the client *drivers* (page 285), there is often minimal or no difference between applications using *replica sets* or standalone instances.

There are two major concepts that *are* important to consider when working with replica sets:

1. *Write Concern* (page 54).

   By default, MongoDB clients receive no response from the server to confirm successful write operations. Most drivers provide a configurable "safe mode," where the server will return a response for all write operations using `getLastError`. For replica sets, *write concern* is configurable to ensure that secondary members of the set have replicated operations before the write returns.

2. *Read Preference* (page 55)

   By default, read operations issued against a replica set return results from the *primary*. Users may configure *read preference* on a per-connection basis to prefer that read operations return on the *secondary* members.

Read preference and *write concern* have particular *consistency* (page 35) implications.

For a more detailed discussion of application concerns, see *Application Development with Replica Sets* (page 54).

5.1.4 Administration and Operations

This section provides a brief overview of concerns relevant to administrators of *replica set* deployments.

For more information on replica set administration, operations, and architecture, see:

- *Replica Set Administration* (page 38)
- *Replication Architectures* (page 51)

Oplog

The *oplog* (operations log) is a special *capped collection* that keeps a rolling record of all operations that modify that 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 replicate this log and apply the operations to themselves in an asynchronous process. All replica set members contain a copy of the oplog, allowing them to maintain the current state of the database. Operations in the oplog are *idempotent*.

By default, the size of the oplog is as follows:

- For 64-bit Linux, Solaris, and FreeBSD systems, MongoDB will allocate 5% of the available free disk space to the oplog.

  If this amount is smaller than a gigabyte, then MongoDB will allocate 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.

Before oplog creation, you can specify the size of your oplog with the `oplogSize` (page 628) option. After you start a replica set member for the first time, you can only change the size of the oplog by using the *Change the Size of the Oplog* (page 85) tutorial.

In most cases, the default oplog size is sufficient. For example, if an oplog that is 5% of free disk space fills up in 24 hours of operations, then secondaries can stop copying entries from the oplog for 24 hours before they require full resyncing. However, most replica sets have much lower operation volumes, and their oplogs can hold a much larger number of operations.

The following factors affect how MongoDB uses space in the oplog:

- Update operations that affect 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 disk utilization.

- If you delete roughly the same amount of data as you insert.

  In this situation the database will not grow significantly in disk utilization, but the size of the operation log can be quite large.

- If a significant portion of your workload entails in-place updates.

  In-place updates create a large number of operations but do not change the quantity data on disk.

If you can predict your replica set's workload to resemble one of the above patterns, then you may want to consider creating an oplog that is larger than the default. Conversely, if the predominance of activity of your MongoDB-based application are reads and you are writing a small amount of data, you may find that you need a much smaller oplog.

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 49).

For additional information about oplog behavior, see *Oplog Internals* (page 62) and *Syncing* (page 64).

Replica Set Deployment

Without replication, a standalone MongoDB instance represents a single point of failure and any disruption of the MongoDB system will render the database unusable and potentially unrecoverable. Replication increase the reliability of the database instance, and replica sets are capable of distributing reads to *secondary* members depending on *read preference*. For database work loads dominated by read operations, (i.e. "read heavy") replica sets can greatly increase the capability of the database system.

The minimum requirements for a replica set include two members with data, for a *primary* and a secondary, and an *arbiters* (page 41). In most circumstances, however, you will want to deploy three data members.

For those deployments that rely heavily on distributing reads to secondary instances, add additional members to the set as load increases. As your deployment grows, consider adding or moving replica set members to secondary data centers or to geographically distinct locations for additional redundancy. While many architectures are possible, always ensure that the quorum of members required to elect a primary remains in your main facility.

Depending on your operational requirements, you may consider adding members configured for a specific purpose including, a *delayed member* to help provide protection against human errors and change control, a *hidden member* to provide an isolated member for reporting and monitoring, and/or a *secondary only member* (page 39) for dedicated backups.

The process of establishing a new replica set member can be resource intensive on existing members. As a result, deploy new members to existing replica sets significantly before current demand saturates the existing members.

Note: *Journaling*, provides single-instance write durability. The journaling greatly improves the reliability and durability of a database. Unless MongoDB runs with journaling, when a MongoDB instance terminates ungracefully, the database can end in a corrupt and unrecoverable state.

You should assume that a database, running without journaling, that suffers a crash or unclean shutdown is in corrupt or inconsistent state.

Use journaling, however, do not forego proper replication because of journaling.

64-bit versions of MongoDB after version 2.0 have journaling enabled by default.

Security

In most cases, *replica set* 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` (page 623) on all members to permit authentication.

For more information, see the *Security Considerations for Replica Sets* (page 46) section in the *Replica Set Administration* (page 38) document.

Architectures

The architecture and design of the *replica set* deployment can have a great impact on the set's capacity and capability. This section provides a general overview of best practices for replica set architectures.

This document provides an overview of the *complete* functionality of replica sets, which highlights the flexibility of the replica set and its configuration. However, for most production deployments a conventional 3-member replica set with `members[n].priority` (page 663) values of `1` are sufficient.

While the additional flexibility discussed is below helpful for managing a variety of operational complexities, it always makes sense to let those complex requirements dictate complex architectures, rather than add unnecessary complexity to your deployment.

Consider the following factors when developing an architecture for your replica set:

- Ensure that the members of the replica set will always be able to elect a *primary*. Run an odd number of members or run an *arbiter* on one of your application servers if you have an even number of members.
- With geographically distributed members, know where the "quorum" of members will be in the case of any network partitions. Attempt to ensure that the set can elect a primary among the members in the primary data center.
- Consider including a *hidden* (page 40) or *delayed member* (page 40) in your replica set to support dedicated functionality, like backups, reporting, and testing.
- Consider keeping one or two members of the set in an off-site data center, but make sure to configure the *priority* (page 34) to prevent it from becoming primary.

For more information regarding replica set configuration and deployments see *Replication Architectures* (page 51).

5.2 Replica Set Administration

Replica sets automate most administrative tasks associated with database replication. Nevertheless, several operations related to deployment and systems management require administrator intervention remain. This document provides an overview of those tasks, in addition to a collection of troubleshooting suggestions for administers of replica sets.

See Also:

- `rs.status()` and `db.isMaster()`
- *Replica Set Reconfiguration Process* (page 665)
- `rs.conf()` and `rs.reconfig()`
- *Replica Set Configuration* (page 661)

The following tutorials provide task-oriented instructions for specific administrative tasks related to replica set operation.

- *Deploy a Replica Set* (page 73)
- *Convert a Standalone to a Replica Set* (page 76)
- *Add Members to a Replica Set* (page 77)
- *Deploy a Geographically Distributed Replica Set* (page 79)
- *Change the Size of the Oplog* (page 85)

- *Force a Member to Become Primary* (page 87)
- *Change Hostnames in a Replica Set* (page 89)
- *Convert a Secondary to an Arbiter* (page 93)
- *Reconfigure a Replica Set with Unavailable Members* (page 95)
- *Recover MongoDB Data following Unexpected Shutdown* (page 193)

5.2.1 Member Configurations

All *replica sets* have a single *primary* and one or more *secondaries*. Replica sets allow you to configure secondary members in a variety of ways. This section describes these configurations.

Note: A replica set can have up to 12 members, but only 7 members can have votes. For configuration information regarding non-voting members, see *Non-Voting Members* (page 42).

Warning: The `rs.reconfig()` shell command can force the current primary to step down, which causes an *election* (page 34). When the primary steps down, the `mongod` closes all client connections. While, this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods. To successfully reconfigure a replica set, a majority of the members must be accessible.

See Also:

The *Elections* (page 34) section in the *Replication Fundamentals* (page 33) document, and the *Election Internals* (page 63) section in the *Replication Internals* (page 61) document.

Secondary-Only Members

The secondary-only configuration prevents a *secondary* member in a *replica set* from ever becoming a *primary* in a *failover*. You can set secondary-only mode for any member of the set.

For example, you may want to configure all members of a replica sets located outside of the main data centers as secondary-only to prevent these members from ever becoming primary.

To configure a member as secondary-only, set its `members[n].priority` (page 663) value to `0`. Any member with a `members[n].priority` (page 663) equal to `0` will never seek *election* (page 34) and cannot become primary in any situation. For more information on priority levels, see *Member Priority* (page 34).

As an example of modifying member priorities, assume a four-member replica set with member `_id` values of: `0`, `1`, `2`, and `3`. Use the following sequence of operations in the `mongo` shell to modify member priorities:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[1].priority = 0.5
cfg.members[2].priority = 1
cfg.members[3].priority = 2
rs.reconfig(cfg)
```

This sets the following:

- Member `0` to a priority of `0` so that it can never become *primary*.
- Member `1` to a priority of `0.5`, which makes it less likely to become primary than other members but doesn't prohibit the possibility.

- Member 2 to a priority of 1, which is the default value. Member 2 becomes primary if no member with a *higher* priority is eligible.
- Member 3 to a priority of 2. Member 3 becomes primary, if eligible, under most circumstances.

Note: If your replica set has an even number of members, add an *arbiter* (page 41) to ensure that members can quickly obtain a majority of votes in an election for primary.

See Also:

members[n].priority (page 663) and *Replica Set Reconfiguration* (page 665).

Hidden Members

Hidden members are part of a replica set but cannot become primary and are invisible to client applications. *However,* hidden members **do** vote in *elections* (page 34).

Hidden members are ideal for instances that will have significantly different usage patterns than the other members and require separation from normal traffic. Typically, hidden members provide reporting, dedicated backups, and dedicated read-only testing and integration support.

Hidden members have members[n].priority (page 663) set 0 and have members[n].hidden (page 663) set to true.

To configure a *hidden member*, use the following sequence of operations in the mongo shell:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
rs.reconfig(cfg)
```

After re-configuring the set, the member with the _id of 0 has a priority of 0 so that it cannot become primary. The other members in the set will not advertise the hidden member in the isMaster or db.isMaster() output.

Note: You must send the rs.reconfig() command to a set member that *can* become *primary*. In the above example, if you issue the rs.reconfig() operation to the member with the _id of 0, the operation fails.

Note: 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 (page 676) to prevent queries from reaching the hidden member.

See Also:

Replica Set Read Preference (page 55) and *Replica Set Reconfiguration* (page 665).

Delayed Members

Delayed members copy and apply operations from the primary's *oplog* with a specified delay. If a member has a delay of one hour, then the latest entry in this member's oplog will not be more recent than one hour old, and the state of data for the member will reflect the state of the set an hour earlier.

Example

If the current time is 09:52 and the secondary is a delayed by an hour, no operation will be more recent than 08:52.

Delayed members may help recover from various kinds of human error. Such errors may include inadvertently deleted databases or botched application upgrades. Consider the following factors when determining the amount of slave delay to apply:

- Ensure that the length of the delay is equal to or greater than your maintenance windows.
- The size of the oplog is sufficient to capture *more than* the number of operations that typically occur in that period of time. For more information on oplog size, see the *Oplog* (page 36) topic in the *Replication Fundamentals* (page 33) document.

Delayed members must have a *priority* set to `0` to prevent them from becoming primary in their replica sets. Also these members should be *hidden* (page 40) to prevent your application from seeing or querying this member.

To configure a *replica set* member with a one hour delay, use the following sequence of operations in the `mongo` shell:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

After the replica set reconfigures, the set member with the `_id` of `0` has a priority of `0` and cannot become *primary*. The `slaveDelay` (page 663) value delays both replication and the member's *oplog* by 3600 seconds (1 hour). Setting `slaveDelay` (page 663) to a non-zero value also sets `hidden` (page 663) to `true` for this replica set so that it does not receive application queries in normal operations.

Warning: The length of the secondary `slaveDelay` (page 663) must fit within the window of the oplog. If the oplog is shorter than the `slaveDelay` (page 663) window, the delayed member cannot successfully replicate operations.

See Also:

`members[n].slaveDelay` (page 663), *Replica Set Reconfiguration* (page 665), *Oplog* (page 36), *Changing Oplog Size* (page 45) in this document, and the *Change the Size of the Oplog* (page 85) tutorial.

Arbiters

Arbiters are special `mongod` instances that do not hold a copy of the data and thus cannot become primary. Arbiters exist solely participate in *elections* (page 34).

Note: Because of their minimal system requirements, you may safely deploy an arbiter on a system with another workload, such as an application server or monitoring member.

Warning: Do not run arbiter processes on a system that is an active *primary* or *secondary* of its *replica set*.

Arbiters never receive the contents of any collection but do have the following interactions with the rest of the replica set:

- Credential exchanges that authenticate the arbiter with the replica set. All MongoDB processes within a replica set use keyfiles. These exchanges are encrypted.

  MongoDB only transmits the authentication credentials in a cryptographically secure exchange, and encrypts no other exchange.

- Exchanges of replica set configuration data and of votes. These are not 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 *Using MongoDB with SSL Connections* (page 167) for more information. As with all MongoDB components, run arbiters on secure networks.

To add an arbiter, see *Adding an Arbiter* (page 44).

Non-Voting Members

You may choose to change the number of votes that each member has in *elections* (page 34) for *primary*. In general, all members should have only 1 vote to prevent intermittent ties, deadlock, or the wrong members from becoming *primary*. Use *replica set priorities* (page 34) to control which members are more likely to become primary.

To disable a member's ability to vote in elections, use the following command sequence in the `mongo` shell.

```
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 set members with the `_id` values of `3`, `4`, and `5`. This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. If you have three non-voting members, you can add three additional voting members to your set. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

Note: 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 *Replica Set Priorities* (page 34) to control which members are more likely to become primary.

See Also:

`members[n].votes` (page 664) and *Replica Set Reconfiguration* (page 665).

5.2.2 Procedures

This section gives overview information on certain procedures. Most procedures, however, are found in the *replica set tutorials* (page 73).

Adding Members

Before adding a new member to an existing *replica set*, do one of the following to prepare the new member's *data directory*:

- 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 after a short interval. Copying the data over manually shortens 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 36). If the difference in the amount of time between the most recent operation and the most

recent operation to the database exceeds the length of the *oplog* on the existing members, then the new instance will have to completely resynchronize, as described in *Resyncing a Member of a Replica Set* (page 45).

Use `db.printReplicationInfo()` to check the current state of replica set members with regards to the oplog.

For the procedure to add a member to a replica set, see *Add Members to a Replica Set* (page 77).

Removing Members

You may remove a member of a replica set at any time; *however*, for best results always *shut down* the `mongod` instance before removing it from a replica set. Changed in version 2.2: Before 2.2, you *had* to shut down the `mongod` instance before removing it. While 2.2 removes this requirement, it remains good practice. To remove a member, use the `rs.remove()` method in the `mongo` shell while connected to the current *primary*. Issue the `db.isMaster()` command when connected to *any* member of the set to determine the current primary. Use a command in either of the following forms to remove the member:

```
rs.remove("mongo2.example.net:27017")
rs.remove("mongo3.example.net")
```

This operation disconnects the shell briefly and forces a re-connection as the *replica set* renegotiates which member will be primary. The shell displays an error even if this command succeeds.

You can re-add a removed member to a replica set at any time using the *procedure for adding replica set members* (page 42). Additionally, consider using the *replica set reconfiguration procedure* (page 665) to change the `members[n].host` (page 662) value to rename a member in a replica set directly.

Replacing a Member

Use this procedure to replace a member of a replica set when the hostname has changed. This procedure preserves all existing configuration for a member, except its hostname/location.

You may need to replace a replica set member if you want to replace an existing system and only need to change the hostname rather than completely replace all configured options related to the previous member.

Use `rs.reconfig()` to change the value of the `members[n].host` (page 662) field to reflect the new hostname or port number. `rs.reconfig()` will not change the value of `members[n]._id` (page 662).

```
cfg = rs.conf()
cfg.members[0].host = "mongo2.example.net:27019"
rs.reconfig(cfg)
```

Warning: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 34). This causes the current shell session, and clients connected to this replica set, to produce an error even when the operation succeeds.

Adjusting Priority

To change the value of the `members[n].priority` (page 663) value 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 `members[n].priority` (page 663) value in the `cfg` document for `members[n]._id` (page 662) of `0`, `1`, or `2`. The final operation calls `rs.reconfig()` with the argument of `cfg` to initialize the new configuration.

If a member has `members[n].priority` (page 663) set to `0`, it is ineligible to become *primary* and will not seek election. *Hidden members* (page 40), *delayed members* (page 40), and *arbiters* (page 41) all have `members[n].priority` (page 663) set to `0`.

All members have a `members[n].priority` (page 663) equal to `1` by default.

The value of `members[n].priority` (page 663) 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 `members[n].priority` (page 663) value is irrelevant.

Replica sets will preferentially elect and maintain the primary status of the member with the highest `members[n].priority` (page 663) 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 665) example revolves around changing the priorities of the `members` of a replica set.

Adding an Arbiter

For a description of *arbiters* and their purpose in *replica sets*, see *Arbiters* (page 41).

To prevent tied *elections*, do not add an arbiter to a set if the set already has an odd number of voting members.

Because arbiters do not hold a copies of collection data, they have minimal resource requirements and do not require dedicated hardware.

1. Create a data directory for the arbiter. The `mongod` uses this directory for configuration information. It *will not* hold database collection data. The following example creates the `http://docs.mongodb.org/manual/data/arb` data directory:

   ```
   mkdir /data/arb
   ```

2. Start the arbiter, making sure to specify the replica set name and the data directory. Consider the following example:

   ```
   mongod --port 30000 --dbpath /data/arb --replSet rs
   ```

3. In a `mongo` shell connected to the *primary*, add the arbiter to the replica set by issuing the `rs.addArb()` method, which uses the following syntax:

   ```
   rs.addArb("<hostname>:<port>")
   ```

   For example, if the arbiter runs on `m1.example.net:30000`, you would issue this command:

```
rs.addArb("m1.example.net:30000")
```

Changing Oplog Size

The following is an overview of the procedure for changing the size of the oplog. For a detailed procedure, see *Change the Size of the Oplog* (page 85).

1. Shut down the current *primary* instance in the *replica set* and then restart it on a different port and in "standalone" mode.
2. Create a backup of the old (current) oplog. This is optional.
3. Save the last entry from the old oplog.
4. Drop the old oplog.
5. Create a new oplog of a different size.
6. Insert the previously saved last entry from the old oplog into the new oplog.
7. Restart the server as a member of the replica set on its usual port.
8. Apply this procedure to any other member of the replica set that *could become* primary.

Resyncing a Member of a Replica Set

When a secondary's replication process falls behind so far that *primary* overwrites oplog entries that the secondary has not yet replicated, that secondary cannot catch up and becomes "stale." When that occurs, you must resync the member by removing its data and replacing it with up-to-date data.

To do so, use one of the following approaches:

- Restart the `mongod` with an empty data directory and let MongoDB's normal replication syncing feature restore the data. This is the more simple option, but may take longer to replace the data.

  See *Automatically Resync a Stale Member* (page 45).

- 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 *Resync by Copying Data from Another Member* (page 46).

Automatically Resync a Stale Member

This procedure relies on MongoDB's regular process for syncing a new member to restore the data on the stale member. For an overview of how MongoDB syncs *replica sets*, see the *Syncing* (page 64) section.

To resync the stale member:

1. Stop the member's `mongod` instance using the *`mongod --shutdown`* (page 586) option. Make sure to set *`--dbpath`* (page 583) to the member's data directory, as in the following:

   ```
   mongod --dbpath /data/db/ --shutdown
   ```

2. Delete all data and sub-directories from the member's data directory such that the directory is empty.
3. Restart the `mongod` instance on the member. Consider the following example:

   ```
   mongod --dbpath /data/db/ --replSet rsProduction
   ```

At this point, the `mongod` will resync. This process may take a long time, depending on the size of the database and speed of the network. Remember that this operation may have an impact on the working set and/or traffic to existing primary other members of the set.

Resync by Copying Data from Another Member

This approach uses the data directory of an existing member to "seed" the stale member. The data must be recent enough to allow the new member to catch up with the *oplog*.

To resync by copying data from another member, use one of the following approaches:

- Create a snapshot of another member's data and then restore that snapshot to the stale member. Use the snapshot procedures in *Backup and Restoration Strategies* (page 180).
- Lock another member's data with the `db.fsyncLock()` command, copy all of the data in the data directory, and then restore the data to the stale member. Use the procedures for backup storage in *Backup and Restoration Strategies* (page 180).

5.2.3 Security Considerations for Replica Sets

In most cases, 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.)

Additionally, MongoDB provides an authentication mechanism for `mongod` and `mongos` (page 676) instances connecting to replica sets. These instances enable authentication but specify a shared key file that serves as a shared password. New in version 1.8: for replica sets (1.9.1 for sharded replica sets) added support for authentication. To enable authentication 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`* (page 583) (or *`mongos --keyFile`* (page 590)) options on the command line.

Setting `keyFile` (page 623) 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` (page 676) instances that connect to the set.

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 753
```

Note: Key file permissions are not checked on Windows systems.

5.2.4 Troubleshooting Replica Sets

This section describes common strategies for troubleshooting *replica sets*.

See Also:

Monitoring Database Systems (page 170).

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 *Replica Set Status Reference* (page 659).

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 Monitoring Service. For more information see the documentation for MMS.

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. You can use *write concern* to prevent 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, the secondaries will not be able to read the oplog fast enough to keep up with changes. Setting some level of write concern can slow the overall progress of the batch but will prevent the secondary from falling too far behind.

  To prevent this, use write concern so that MongoDB will perform a safe write (i.e. call `getLastError`) after every 100, 1,000, or other designated number of operations. This provides an opportunity for secondaries to catch up with the primary. Using safe writes, even in batches, can impact write throughout; however, calling `getLastError` will prevents the secondaries from falling too far behind the primary.

  For more information see:

  - *Write Concern* (page 54).
  - The *Oplog* (page 36) topic in the *Replication Fundamentals* (page 33) document.
  - The *Oplog Internals* (page 62) topic in the *Replication Internals* (page 61) document.
  - The *Changing Oplog Size* (page 45) topic in this document.
  - The *Change the Size of the Oplog* (page 85) tutorial.

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 typologies 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 `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.

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:

- The *Oplog* (page 36) topic in the *Replication Fundamentals* (page 33) document.
- The *Delayed Members* (page 40) topic in this document.
- The *Check the Replication Lag* (page 47) topic in this document.

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 *Changing Oplog Size* (page 45) in this document or see the *Change the Size of the Oplog* (page 85) tutorial.

Failover and Recovery

Replica sets feature automated failover. If the *primary* goes offline or becomes unresponsive and a majority of the original set members can still connect to each other, the set will elect a new primary.

While *failover* is automatic, *replica set* administrators should still understand exactly how this process works. This section below describe failover in detail.

In most cases, failover occurs without administrator intervention seconds after the *primary* either steps down, becomes inaccessible, or becomes otherwise ineligible to act as primary. If your MongoDB deployment does not failover according to expectations, consider the following operational errors:

- No remaining member is able to form a majority. This can happen as a result of network partitions that render some members inaccessible. Design your deployment to ensure that a majority of set members can elect a primary in the same facility as core application systems.
- No member is eligible to become primary. Members must have a `members[n].priority` (page 663) setting greater than `0`, have a state that is less than ten seconds behind the last operation to the *replica set*, and generally be *more* up to date than the voting members.

In many senses, *rollbacks* (page 35) represent a graceful recovery from an impossible failover and recovery situation.

Rollbacks occur when a primary accepts writes that other members of the set do not successfully replicate before the primary steps down. When the former primary begins replicating again it performs a "rollback." Rollbacks remove those operations from the instance that were never replicated to the set so that the data set is in a consistent state. The `mongod` program writes rolled back data to a *BSON* file that you can view using `bsondump`, applied manually using `mongorestore`.

You can prevent rollbacks by ensuring safe writes by using the appropriate *write concern*.

See Also:

The *Elections* (page 34) section in the *Replication Fundamentals* (page 33) document, and the *Election Internals* (page 63) section in the *Replication Internals* (page 61) document.

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?
```

The most often cause of this error is wrongly typed value for the `ts` field in the last *oplog* entry might be of the wrong data type. The correct data type is Timestamp.

You can check the data type by running the following two queries against the oplog. If the data is properly typed, the queries will return the same document, otherwise these queries will return different documents. These queries are:

```
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 for type 17 `BSON`, which 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" : "", "
```

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" : "",
```

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.

5.3 Replication Architectures

There is no single ideal *replica set* architecture for every deployment or environment. Indeed the flexibility of replica sets might be their greatest strength. This document describes the most commonly used deployment patterns for replica sets. The descriptions are necessarily not mutually exclusive, and you can combine features of each architecture in your own deployment.

For an overview of operational practices and background information, see the *Architectures* (page 38) topic in the *Replication Fundamentals* (page 33) document.

5.3.1 Three Member Sets

The minimum *recommended* architecture for a replica set consists of:

- One *primary* and
- Two *secondary* members, either of which can become the primary at any time.

  This makes *failover* (page 34) possible and ensures there exists two full and independent copies of the data set at all times. If the primary fails, the replica set elects another member as primary and continues replication until the primary recovers.

Note: While not *recommended*, the minimum *supported* configuration for replica sets includes one *primary*, one *secondary*, and one *arbiter* (page 41). The arbiter requires fewer resources and lowers costs but sacrifices operational flexibility and redundancy.

See Also:

Deploy a Replica Set (page 73).

5.3.2 Sets with Four or More Members

To increase redundancy or to provide additional resources for distributing secondary read operations, you can add additional members to a replica set.

When adding additional members, ensure the following architectural conditions are true:

- The set has an odd number of voting members.

  If you have an *even* number of voting members, deploy an *arbiter* (page 41) to create an odd number.

- The set has no more than 7 voting members at a time.

- Members that cannot function as primaries in a *failover* have their `priority` (page 663) values set to `0`.

  If a member cannot function as a primary because of resource or network latency constraints a `priority` (page 663) value of `0` prevents it from being a primary. Any member with a `priority` value greater than `0` is available to be a primary.

- A majority of the set's members operate in the main data center.

See Also:

Add Members to a Replica Set (page 77).

5.3.3 Geographically Distributed Sets

A geographically distributed replica set provides data recovery should one data center fail. These sets include at least one member in a secondary data center. The member has its the `priority` (page 663) *set* (page 665) to `0` to prevent the member from ever becoming primary.

In many circumstances, these deployments consist of the following:

- One *primary* in the first (i.e., primary) data center.

- One *secondary* member in the primary data center. This member can become the primary member at any time.

- One secondary member in a secondary data center. This member is ineligible to become primary. Set its `members[n].priority` (page 663) to `0`.

If the primary is unavailable, the replica set will elect a new primary from the primary data center.

If the *connection* between the primary and secondary data centers fails, the member in the secondary center cannot independently become the primary.

If the primary data center fails, you can manually recover the data set from the secondary data center. With proper *write concern* there will be no data loss and downtime can be minimal.

When you add a secondary data center, make sure to keep an odd number of members overall to prevent ties during elections for primary by deploying an *arbiter* (page 41) in your primary data center. For example, if you have three members in the primary data center and add a member in a secondary center, you create an even number. To create an odd number and prevent ties, deploy an *arbiter* (page 41) in your primary data center.

See Also:

Deploy a Geographically Distributed Replica Set (page 79)

5.3.4 Non-Production Members

In some cases it may be useful to maintain a member that has an always up-to-date copy of the entire data set but that cannot become primary. You might create such a member to provide backups, to support reporting operations, or to act as a cold standby. Such members fall into one or more of the following categories:

- **Low-Priority**: These members have `members[n].priority` (page 663) settings such that they are either unable to become *primary* or *very* unlikely to become primary. In all other respects these low-priority members are identical to other replica set member. (See: *Secondary-Only Members* (page 39).)
- **Hidden**: These members cannot become primary *and* the set excludes them from the output of `db.isMaster()` and from the output of the database command `isMaster`. Excluding hidden members from such outputs prevents clients and drivers from using hidden members for secondary reads. (See: *Hidden Members* (page 40).)
- **Voting**: This changes the number of votes that a member of the replica set has in elections. In general, use priority to control the outcome of elections, as weighting votes introduces operational complexities and risks. Only modify the number of votes when you need to have more than 7 members in a replica set. (See: *Non-Voting Members* (page 42).)

Note: All members of a replica set vote in elections *except* for *non-voting* (page 42) members. Priority, hidden, or delayed status does not affect a member's ability to vote in an election.

Backups

For some deployments, keeping a replica set member for dedicated backup purposes is operationally advantageous. Ensure this member is close, from a networking perspective, to the primary or likely primary. Ensure that the *replication lag* is minimal or non-existent. To create a dedicated *hidden member* (page 40) for the purpose of creating backups.

If this member runs with journaling enabled, you can safely use standard *block level backup methods* (page 181) to create a backup of this member. Otherwise, if your underlying system does not support snapshots, you can connect `mongodump` to create a backup directly from the secondary member. In these cases, use the `--oplog` (page 598) option to ensure a consistent point-in-time dump of the database state.

See Also:

Backup and Restoration Strategies (page 180).

Delayed Replication

Delayed members are special `mongod` instances in a *replica set* that apply operations from the *oplog* on a delay to provide a running "historical" snapshot of the data set, or a rolling backup. Typically these members provide protection against human error, such as unintentionally deleted databases and collections or failed application upgrades or migrations.

Otherwise, delayed member function identically to *secondary* members, with the following operational differences: they are not eligible for election to primary and do not receive secondary queries. Delayed members *do* vote in *elections* for primary.

See *Replica Set Delayed Nodes* (page 40) for more information about configuring delayed replica set members.

Reporting

Typically *hidden members* provide a substrate for reporting purposes, because the replica set segregates these instances from the cluster. Since no secondary reads reach hidden members, they receive no traffic beyond what replication requires. While hidden members are not electable as primary, they are still able to *vote* in elections for primary. If your operational parameters requires this kind of reporting functionality, see *Hidden Replica Set Nodes* (page 40) and `members[n].hidden` (page 663) for more information regarding this functionality.

Cold Standbys

For some sets, it may not be possible to initialize a new members in a reasonable amount of time. In these situations, it may be useful to maintain a secondary member with an up-to-date copy for the purpose of replacing another member in the replica set. In most cases, these members can be ordinary members of the replica set, but in large sets, with varied hardware availability, or given some patterns of *geographical distribution* (page 52), you may want to use a member with a different *priority*, *hidden*, or voting status.

Cold standbys may be valuable when your *primary* and "hot standby" *secondaries* members have a different hardware specification or connect via a different network than the main set. In these cases, deploy members with *priority* equal to `0` to ensure that they will never become primary. These members will vote in elections for primary but will never be eligible for election to primary. Consider likely failover scenarios, such as inter-site network partitions, and ensure there will be members eligible for election as primary *and* a quorum of voting members in the main facility.

Note: If your set already has `7` members, set the `members[n].votes` (page 664) value to `0` for these members, so that they won't vote in elections.

See Also:

Secondary Only (page 39), and *Hidden Nodes* (page 40).

5.3.5 Arbiters

Deploy an *arbiter* to ensure that a replica set will have a sufficient number of members to elect a *primary*. While having replica sets with 2 members is not recommended for production environments, if you have just two members, deploy an arbiter. Also, for *any replica set with an even number of members*, deploy an arbiter.

To deploy an arbiter, see the *Arbiters* (page 41) topic in the *Replica Set Administration* (page 38) document.

5.4 Application Development with Replica Sets

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. [2] This document describes those options and their implications.

5.4.1 Write Concern

When a *client* sends a write operation to a database server, the operation returns without waiting for the operation to succeed or complete by default. To check if write operations have succeeded, use the `getLastError` command. `getLastError` supports the following options that allow you to tailor the level of "write concern" provided by the command's return or acknowledgment:

- no options.

  Confirms that the `mongod` instance received the write operations. When your application receives this response, the `mongod` instance has committed the write operation to the in-memory representation of the database. This provides a simple and low-latency level of write concern and will allow your application to detect situations where the `mongod` instance becomes inaccessible or insertion errors caused by *duplicate key errors* (page 234).

[2] *Sharded clusters* where the shards are also replica sets provide the same configuration options with regards to write and read operations.

- `j` or "journal" option.

  Confirms the above, and that the `mongod` instance has written the data to the on-disk journal. This ensures that the data is durable if `mongod` or the server itself crashes or shuts down unexpectedly.

- `fsync` option. Deprecated since version 1.8. Do not use the `fsync` option. Confirms that the `mongod` has flushed the in-memory representation of the data to the disk. Instead, use the `j` option to ensure durability.

- `w` option. Only use with replica sets.

  Confirms that the write operation has replicated to the specified number of replica set members. You may specify a specific number of servers *or* specify `majority` to ensure that the write propagates to a majority of set members. The default value of `w` is `1`.

You may combine multiple options, such as `j` and `w`, into a single `getLastError` operation.

Many drivers have a "safe" mode or "write concern" that automatically issues `getLastError` after write operations to ensure the operations complete. Safe mode provides confirmation of write operations, but safe writes can take longer to return and are not required in all applications. Consider the following operations:

```
db.runCommand( { getLastError: 1, w: "majority" } )
db.getLastErrorObj("majority")
```

These equivalent `getLastError` operations ensure that write operations return only after a write operation has replicated to a majority of the members of the set.

Note: If you specify a `w` value greater than the number of available non-*arbiter* replica set members, the operation will block until those members become available. This could cause the operation to block forever. To specify a timeout threshold for the `getLastError` operation, use the `wtimeout` argument.

You can also configure your own "default" `getLastError` behavior for the replica set. Use the `settings.getLastErrorDefaults` (page 664) setting in the *replica set configuration* (page 661). For instance:

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = {w: "majority", j: true}
rs.reconfig(cfg)
```

When the new configuration is active, the `getLastError` operation waits for the write operation to complete on a majority of the set members before returning. Specifying `j: true` makes `getLastError` wait for a complete commit of the operations to the journal before returning.

The `getLastErrorDefaults` setting only affects `getLastError` commands with *no* other arguments.

Note: Use of inappropriate write concern can lead to *rollbacks* (page 35) in the case of *replica set failover* (page 34). Always ensure that your operations have specified the required write concern for your application.

5.4.2 Read Preference

Read preference describes how MongoDB clients route read operations to *secondary* members of a *replica set*.

Background

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, for an application that does not

require fully up-to-date data, you can improve read throughput by distributing some or all reads to secondary members of the replica set.

The following are use cases where you might use secondary reads:

- Running systems operations that do not affect the front-end application, operations such as backups and reports.
- Providing low-latency queries for geographically distributed deployments. If one secondary is closer to an application server than the primary, you may see better performance for that application if you use secondary reads.
- Providing graceful degradation in *failover* (page 34) situations where a set has *no* primary for 10 seconds or more. In this use case, you should give the application the `primaryPreferred` (page 57) read preference, which prevents the application from performing reads if the set has no primary.

MongoDB *drivers* allow client applications to configure a *read preference* on a per-connection, per-collection, or per-operation basis. For more information about secondary read operations in the `mongo` shell, see the `readPref()` method. For more information about a driver's read preference configuration, see the appropriate *Drivers* (page 285) API documentation.

Note: Read preferences affect how an application selects which member to use for read operations. As a result read preferences dictate if the application receives stale or current data from MongoDB. Use appropriate *write concern* (page 54) policies to ensure proper data replication and constancy.

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 107) provides better support for larger scale operations, as clusters can distribute read and write operations across a group of machines.

Read Preference Modes

New in version 2.2. MongoDB *drivers* (page 285) support five read preference modes:

- `primary` (page 56)
- `primaryPreferred` (page 57)
- `secondary` (page 57)
- `secondaryPreferred` (page 57)
- `nearest` (page 57)

You can specify a read preference mode on a per-collection or per-operation basis. The syntax for specifying the read preference mode is specific to the driver and to the idioms of the host language.

Read preference modes are also available to clients connecting to a *sharded cluster* through a `mongos` (page 676). The `mongos` (page 676) 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.

Warning: All read preference modes except `primary` (page 56) may return stale data as *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 56) mode.

For more information, see *read preference background* (page 55) and *read preference behavior* (page 59). See also the documentation for your driver.

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.

`primary` (page 56) read preference modes are not compatible with read preferences mode that use *tag sets* (page 58). If you specify a tag set with `primary` (page 56), the driver produces 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 58), 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 57) mode may return stale data in some situations.

> **Warning:** Changed in version 2.2: `mongos` (page 676) added full support for read preferences. When connecting to a `mongos` (page 676) instance older than 2.2, using a client that supports read preference modes, `primaryPreferred` (page 57) 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 58), 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 60). If no secondaries have matching tags, the read operation produces an error. [3]

Read operations using the `secondary` (page 57) 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 58), 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 60). If no secondaries have matching tags, the read operation produces an error.

Read operations using the `secondaryPreferred` (page 57) mode may return stale data.

nearest

The driver reads from the *nearest* member of the *set* according to the *member selection* (page 60) process. Reads in the `nearest` (page 57) mode do not consider the member's *type*. Reads in `nearest` (page 57) 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 58), 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 60).

[3] If your set has more than one secondary, and you use the `secondary` (page 57) read preference mode, consider the following effect. If you have a *three member replica set* (page 51) with a primary and two secondaries, and if one secondary becomes unavailable, all `secondary` (page 57) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.

Read operations using the `nearest` (page 57) 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 57) mode prefers low latency reads over a member's *primary* or *secondary* status.

For `nearest` (page 57), 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 60) for more information.

Tag Sets

Tag sets allow you to specify custom *read preferences* (page 55) so that your application can target read operations to specific members, based on custom parameters. A tag set for a read operation may resemble the following:

```
{ "disk": "ssd", "use": "reporting" }
```

To fulfill the request, a member would need to have both of these tag sets. Therefore the following tag sets, would satisfy this requirement:

```
{ "disk": "ssd", "use": "reporting" }
{ "disk": "ssd", "use": "reporting", "rack": 1 }
{ "disk": "ssd", "use": "reporting", "rack": 4 }
{ "disk": "ssd", "use": "reporting", "mem": "64"}
```

However, 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": 3 }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Therefore, tag sets make it possible to ensure that read operations target specific members in a particular data center or `mongod` instances designated for a particular class of operations, such as reporting or analytics. For information on configuring tag sets, see *Tag Sets* (page 666) in the *Replica Set Configuration* (page 661) document. You can specify tag sets with the following read preference modes:

- `primaryPreferred` (page 57)
- `secondary` (page 57)
- `secondaryPreferred` (page 57)
- `nearest` (page 57)

You cannot specify tag sets with the `primary` (page 56) read preference mode.

Tags are not compatible with `primary` (page 56) and only apply when *selecting* (page 60) a *secondary* member of a set for a read operation. However, the `nearest` (page 57) 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 60) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For more information on how read preferences *modes* (page 56) interact with tag sets, see the documentation for each read preference mode.

Behavior

Changed in version 2.2.

Auto-Retry

Connection 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` (page 676):

- 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 56), if 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 56) and *tag set* (page 58). 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`.
- After detecting a failover situation, [4] the driver attempts to refresh the state of the replica set as quickly as possible.

Request Association

Reads from *secondary* may reflect the state of the data set at different points in time because *secondary* members of a *replica set* may lag behind the current state of the primary by different amounts. To prevent subsequent reads from jumping around in time, the driver can associate application threads to a specific member of the set after the first read. The thread will continue to read from the same member until:

- The application performs a read with a different read preference.
- The thread terminates.
- 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 59), which may be transparent to the application.

If an application thread issues a query with the `primaryPreferred` (page 57) mode while the primary is inaccessible, the thread will carry the association with that secondary for the lifetime of the thread. The thread will associate with the primary, if available, only after issuing a query with a different read preference, even if a primary becomes available. By extension, if a thread issues a read with the `secondaryPreferred` (page 57) when all secondaries

[4] 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*.

are down, it will carry an association with the primary. This application thread will continue to read from the primary even if a secondary becomes available later in the thread's lifetime.

Member Selection

Both clients, by way of their drivers, and mongos (page 676) instances for sharded clusters send periodic "ping," messages to all member of the replica set to determine latency from the application to each mongod instance.

For any operation that targets a member *other* than the *primary*, the driver:

1. Assembles a list of suitable members, taking into account member type (i.e. secondary, primary, or all members.)
2. Determines which suitable member is the closest to the client in absolute terms.
3. Builds a list of members that are within a defined ping distance (in milliseconds) of the "absolute nearest" member. [5]
4. Selects a member from these hosts at random. The member receives the read operation.

Once the application selects a member of the set to use for read operations, the driver continues to use this connection for read preference until the application specifies a new read preference or something interrupts the connection. See *Request Association* (page 59) for more information.

Sharding and mongos

Changed in version 2.2: Before version 2.2, mongos (page 676) did not support the *read preference mode semantics* (page 56). In most *sharded clusters*, a *replica set* provides each shard where read preferences are also applicable. Read operations in a sharded cluster, with regard to read preference, are identical to unsharded replica sets.

Unlike simple replica sets, in sharded clusters, all interactions with the shards pass from the clients to the mongos (page 676) instances that are actually connected to the set members. mongos (page 676) is responsible for the application of the 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 (page 676) is at least version 2.2. All mongos (page 676) maintain their own connection pool to the replica set members. As a result:

- A request without a specified preference has primary (page 56), the default, unless, the mongos (page 676) reuses an existing connection that has a different mode set.

  Always explicitly set your read preference mode to prevent confusion.

- All nearest (page 57) and latency calculations reflect the connection between the mongos (page 676) and the mongod instances, not the client and the mongod instances.

  This produces the desired result, because all results must pass through the mongos (page 676) before returning to the client.

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 56) for the following commands:

- group

[5] Applications can configure the threshold used in this stage. The default "acceptable latency" is 15 milliseconds. For mongos (page 676) you can use the *--localThreshold* (page 591) or localThreshold (page 630) runtime options to set this value.

- mapReduce [6]
- aggregate
- collStats
- dbStats
- count
- distinct
- geoNear
- geoSearch
- geoWalk

Uses for non-Primary Read Preferences

You must exercise care when specifying read preference: modes other than primary (page 56) can *and will* return stale data. These secondary queries will not include most recent write operations to the replica set's *primary*. Nevertheless, there are several common use cases for using non-primary (page 56) read preference modes:

- Reporting and analytics workloads.

  Having these queries target a *secondary* helps distribute load and prevent these operations from affecting the primary workload of the primary.

  Also consider using secondary (page 57) in conjunction with a direct connection to a *hidden member* (page 40) of the set.

- 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 52) and using a non primary read preference or the nearest (page 57) to avoid network latency.

Using read modes other than primary (page 56) and primaryPreferred (page 57) to provide extra capacity is not in and of itself justification for non-primary (page 56) in many cases. Furthermore, *sharding* (page 105) increases read and write capacity by distributing read and write operations across a group of machines.

5.5 Replication Internals

This document provides a more in-depth explanation of the internals and operation of *replica set* features. This material is not necessary for normal operation or application development but may be useful for troubleshooting and for further understanding MongoDB's behavior and approach.

For additional information about the internals of replication replica sets see the following resources in the MongoDB Manual:

- *Local Database* (page 677)
- *Replica Set Commands* (page 64)
- *Replication Info Reference* (page 667)
- *Replica Set Configuration* (page 661)

[6] Only "inline" mapReduce operations that do not write data support read preference, otherwise these operations must run on the *primary* members.

5.5.1 Oplog Internals

For an explanation of the oplog, see *Oplog* (page 36).

Under various exceptional situations, updates to a *secondary's* oplog might lag behind the desired performance time. See *Replication Lag* (page 47) for details.

All members of a *replica set* send heartbeats (pings) to all other members in the set and can import operations to the local oplog from any other member in the set.

Replica set oplog operations are *idempotent*. The following operations require idempotency:

- initial sync
- post-rollback catch-up
- sharding chunk migrations

5.5.2 Data Integrity

Read Preference Internals

MongoDB uses *single-master replication* to ensure that the database remains consistent. However, clients may modify the *read preferences* (page 55) on a per-connection basis in order to distribute read operations to the *secondary* members of a *replica set*. Read-heavy deployments may achieve greater query throughput by distributing reads to secondary members. But keep in mind that replication is asynchronous; therefore, reads from secondaries may not always reflect the latest writes to the *primary*.

See Also:

Consistency (page 35)

Note: Use `db.getReplicationInfo()` from a secondary member and the *replication status* (page 667) output to asses the current state of replication and determine if there is any unintended replication delay.

5.5.3 Member Configurations

Replica sets can include members with the following four special configurations that affect membership behavior:

- *Secondary-only* (page 39) members have their `priority` (page 663) values set to `0` and thus are not eligible for election as primaries.
- *Hidden* (page 40) members do not appear in the output of `db.isMaster()`. This prevents clients from discovering and potentially querying the member in question.
- *Delayed* (page 40) members lag a fixed period of time behind the primary. These members are typically used for disaster recovery scenarios. For example, if an administrator mistakenly truncates a collection, and you discover the mistake within the lag window, then you can manually fail over to the delayed member.
- *Arbiters* (page 41) exist solely to participate in elections. They do not replicate data from the primary.

In almost every case, replica sets simplify the process of administering database replication. However, replica sets still have a unique set of administrative requirements and concerns. Choosing the right *system architecture* (page 51) for your data set is crucial.

See Also:

The *Member Configurations* (page 39) topic in the *Replica Set Administration* (page 38) document.

5.5.4 Security Internals

Administrators of replica sets also have unique *monitoring* (page 175) and *security* (page 46) concerns. The *replica set functions* in the `mongo` shell, provide the tools necessary for replica set administration. In particular use the `rs.conf()` to return a *document* that holds the *replica set configuration* (page 661) and use `rs.reconfig()` to modify the configuration of an existing replica set.

5.5.5 Election Internals

Elections are the process *replica set* members use to select which member should become *primary*. A primary is the only member in the replica set that can accept write operations, including `insert()`, `update()`, and `remove()`.

The following events can trigger an election:

- You initialize a replica set for the first time.
- A primary steps down. A primary will step down in response to the `replSetStepDown` (page 519) command or if it sees that one of the current secondaries is eligible for election *and* has a higher priority. A primary also will step down when it cannot contact a majority of the members of the replica set. When the current primary steps down, it closes all open client connections to prevent clients from unknowingly writing data to a non-primary member.
- A *secondary* member loses contact with a primary. A secondary will call for an election if it cannot establish a connection to a primary.
- A *failover* occurs.

In an election, all members have one vote, including *hidden* (page 40) members, *arbiters* (page 41), and even recovering members. Any `mongod` can veto an election.

In the default configuration, all members have an equal chance of becoming primary; however, it's possible to set `priority` (page 663) values that weight the election. In some architectures, there may be operational reasons for increasing the likelihood of a specific replica set member becoming primary. For instance, a member located in a remote data center should *not* become primary. See: *Member Priority* (page 34) for more information.

Any member of a replica set can veto an election, even if the member is a *non-voting member* (page 42).

A member of the set will veto an election under the following conditions:

- 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 *secondary only member* (page 39) [7] is the most current member at the time of the election, 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 member has more recent operations (i.e. a higher "optime") than the member seeking election, from the perspective of the voting member.
- The current primary will veto an election if it has the same or more recent operations (i.e. a "higher or equal optime") than the member seeking election.

The first member to receive votes from a majority of members in a set becomes the next primary until the next election. Be aware of the following conditions and possible situations:

- Replica set members send heartbeats (pings) to each other every 2 seconds. If a heartbeat does not return for more than 10 seconds, the other members mark the delinquent member as inaccessible.

[7] Remember that *hidden* (page 40) and *delayed* (page 40) imply *secondary-only* (page 39) configuration.

- Replica set members compare priorities only with other members of the set. The absolute value of priorities does not have any impact on the outcome of replica set elections, with the exception of the value `0`, which indicates the member cannot become primary and cannot seek election. For details, see *Adjusting Priority* (page 43).
- A replica set member cannot become primary *unless* it has the highest "optime" of any visible member in the set.
- If the member of the set with the highest priority is within 10 seconds of the latest *oplog* entry, then the set will *not* elect a primary until the member with the highest priority catches up to the latest operation.

See Also:

Non-voting members in a replica set (page 42), *Adjusting Priority* (page 43), and `replica configuration` (page 664).

Elections and Network Partitions

Members on either side of a network partition cannot see each other when determining whether a majority is available to hold an election.

That means that if a primary steps down and neither side of the partition has a majority on its own, the set will not elect a new primary and the set will become read only. The best practice is to have and a majority of servers in one data center and one server in another.

5.5.6 Syncing

In order to remain up-to-date with the current state of the *replica set*, set members sync, or copy, *oplog* entries from other members.

When a new member joins a set or an existing member restarts, the member waits to receive heartbeats from other members. By default, the member syncs from the *the closest* member of the set that is either the primary or another secondary with more recent oplog entries. This prevents two secondaries from syncing from each other.

In version 2.0, secondaries only change sync targets if the connection between secondaries drops or produces an error.

For example:

1. If you have two secondary members in one data center and a primary in a second facility, and if you start all three instances at roughly the same time (i.e. with no existing data sets or oplog,) both secondaries will likely sync from the primary, as neither secondary has more recent oplog entries.

   If you restart one of the secondaries, then when it rejoins the set it will likely begin syncing from the other secondary, because of proximity.

2. If you have a primary in one facility and a secondary in an alternate facility, and if you add another secondary to the alternate facility, the new secondary will likely sync from the existing secondary because it is closer than the primary.

See Also:

Resyncing a Member of a Replica Set (page 45)

5.6 Replica Set Commands

This reference collects documentation for all *JavaScript methods* (page 65) for the `mongo` shell that support *replica set* functionality, as well as all *database commands* (page 68) related to replication function.

See *Replication* (page 31), for a list of all replica set documentation.

5.6.1 JavaScript Methods

The following methods apply to replica sets. For a complete list of all methods, see *JavaScript Methods* (page 527).

`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*.

See Also:

"*Replica Set Status Reference* (page 659)" for documentation of this output.

`db.isMaster`

Returns a status document with fields that includes the `ismaster` field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* `isMaster`

`rs.initiate`

Parameters

- **configuration** – Optional. A *document* that specifies the configuration of a replica set. If not specified, MongoDB will use a default configuration.

Initiates a replica set. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set. Consider the following model of the most basic configuration for a 3-member replica set:

```
{
    _id : <setname>,
     members : [
         {_id : 0, host : <host0>},
         {_id : 1, host : <host1>},
         {_id : 2, host : <host2>},
     ]
}
```

This function provides a wrapper around the "`replSetInitiate`" *database command*.

`rs.conf`

Returns a *document* that contains the current *replica set* configuration object.

`rs.config`

`rs.config()` is an alias of `rs.conf()`.

`rs.reconfig`

Parameters

- **configuration** – A *document* that specifies the configuration of a replica set.
- **force** – Optional. Specify `{ force: true }` as the force parameter to force 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 *rollback* situations.

Initializes a new *replica set* configuration. This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.reconfig()` provides a wrapper around the "`replSetReconfig`" *database command*.

`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.

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 isn't connected to the current member, or you're 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 661)" and "*Replica Set Administration* (page 38)".

`rs.add`

Specify one of the following forms:

Parameters

- **host** (*string*) – Either a string or a document. If a string, specifies a host (and optionally port-number) for a new host member for the replica set; MongoDB will add this host with the default configuration. If a document, specifies any attributes about a member of a replica set.
- **arbiterOnly** (*Boolean*) – Optional. If `true`, this host is an arbiter. If the second argument evaluates to `true`, as is the case with some *documents*, then this instance will become an arbiter.

Provides a simple method to add a member to an existing *replica set*. You can 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 78) example.
2. as a configuration *document*, as in the *Add a Member to an Existing Replica Set (Alternate Procedure)* (page 79) example.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node 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 661) document for full documentation of all replica set configuration options.

`rs.addArb`

Parameters

- **host** (*string*) – Specifies a host (and optionally port-number) for a arbiter member for the replica set.

Adds a new *arbiter* to an existing replica set.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.stepDown`

Parameters

- **seconds** (*init*) – Specify the duration of this operation. If not specified the command uses the default value of 60 seconds.

Returns disconnects shell.

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 node is not primary.

This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which node 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` (page 519).

`rs.freeze`

Parameters

- **seconds** (*init*) – Specify the duration of this operation.

Forces the current node to become ineligible to become primary for the period specified.

`rs.freeze()` provides a wrapper around the *database command* `replSetFreeze`.

`rs.remove`

Parameters

- **hostname** – Specify one of the existing hosts to remove from the current replica set.

Removes the node 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 negotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

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* nodes. See the `readPref()` method for more fine-grained control over *read preference* (page 55) in the `mongo` shell.

`db.isMaster`

Returns a status document with fields that includes the `ismaster` field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* `isMaster`

`rs.help`
Returns a basic help text for all of the *replication* (page 33) 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 sync from in the form of `[hostname]:[port]`.

See `replSetSyncFrom` for more details.

5.6.2 Database Commands

The following commands apply to replica sets. For a complete list of all commands, see `http://docs.mongodb.org/manual/reference/commands`.

isMaster
The `isMaster` command provides a basic overview of the current replication configuration. MongoDB *drivers* and *clients* use this command to determine what kind of member they're connected to and to discover additional members of a *replica set*. The `db.isMaster()` method provides a wrapper around this database command.

The command takes the following form:

```
{ isMaster: 1 }
```

This command returns a *document* containing the following fields:

`isMaster.setname`
The name of the current replica set, if applicable.

`isMaster.ismaster`
A boolean value that reports when this node is writable. If `true`, then the current node is either a *primary* node in a *replica set*, a *master* node in a master-slave configuration, of a standalone `mongod`.

`isMaster.secondary`
A boolean value that, when `true`, indicates that the current node is a *secondary* member of a *replica set*.

`isMaster.hosts`
An array of strings in the format of "`[hostname]:[port]`" listing all nodes in the *replica set* that are not "*hidden*".

`isMaster.primary`
The `[hostname]:[port]` for the current *replica set primary*, if applicable.

`isMaster.me`
The `[hostname]:[port]` of the node responding to this command.

`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.localTime`
New in version 2.1.1. Returns the local server time in UTC. This value is a *ISOdate*. You can use the `toString()` JavaScript method to convert this value to a local date string, as in the following example:

```
db.isMaster().localTime.toString();
```

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 no 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` (page 519) 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 the it against the *admin database*.

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 }
```

However, you can also run this command from the shell like so:

```
rs.status()
```

See Also:

"*Replica Set Status Reference* (page 659)" and "*Replication Fundamentals* (page 33)"

replSetInitiate

The `replSetInitiate` command initializes a new replica set. Use the following syntax:

```
{ replSetInitiate : <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 661)," "*Replica Set Administration* (page 38)," and "*Replica Set Reconfiguration* (page 665)."

replSetMaintenance

The `replSetMaintenance` 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: 1`, 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.

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 :dbcommand‘replSetReconfig‘ operation from occurring at the same time.

`replSetSyncFrom`

New in version 2.2.

Options

- **host** – Specifies the name and port number of the set member that you want *this* member to sync from. Use the `[hostname]:[port]` form.

`replSetSyncFrom` allows you to explicitly configure which host the current `mongod` will poll *oplog* entries from. This operation may be useful for testing different patterns and in situations where a set member is not syncing from the host you want. You may **not** use this command to force a member to sync from:

- itself.
- an arbiter.
- 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 sync from a member that is more than 10 seconds behind the current member, `mongod` will return and log a warning, but *will* sync from such members.

The command has the following prototype form:

```
{ replSetSyncFrom: "[hostname]:[port]" }
```

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` provides a temporary override of default behavior. When you restart the `mongod` instance, it will revert to the default syncing logic. Always exercise caution with `replSetSyncFrom` when overriding the default behavior.

CHAPTER

SIX

TUTORIALS

The following tutorials describe certain replica set maintenance operations in detail:

6.1 Deploy a Replica Set

This tutorial describes how to create a three member *replica set* from three existing instances of MongoDB. The tutorial provides one procedure for development and test systems and a separate procedure for production systems.

To deploy a replica set from a single standalone MongoDB instance, see *Convert a Standalone to a Replica Set* (page 76).

For background information on replica set deployments, see *Replication Fundamentals* (page 33) and *Replication Architectures* (page 51).

6.1.1 Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. Additionally, these sets have sufficient capacity for many distributed read operations. Most deployments require no additional members or configuration.

6.1.2 Requirements

A replica set requires three distinct systems so that each system can run its own instance of `mongod`. For development systems you can run all three instances of the `mongod` process on a local system. (e.g. a laptop) or within a virtual instance. For production environments, you should endeavor to maintain as much separation between the members as possible. For example, when using VMs in Production, each member should live on separate host servers, served by redundant power circuits, and with redundant network paths.

6.1.3 Procedure

These procedures assume you already have instances of MongoDB installed on the systems you will add as members of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials* (page 9).

Deploy a Development or Testing Replica Set

The examples in this procedure create a new replica set named `rs0`.

1. Before creating your replica set, verify that every member can successfully connect to every other member. The network configuration must allow all possible connections between any any two members. To test connectivity, see *Test Connections Between all Members* (page 48).

2. Start three instances of `mongod` as members of a replica set named `rs0`, as described in this step. For ephemeral tests and the purposes of this guide, you may run the `mongod` instances in separate windows of GNU Screen. OS X and most Linux distributions come with screen installed by default [1] systems.

   (a) Create the necessary data directories by issuing a command similar to the following:

   ```
   mkdir -p /srv/mongodb/rs0-0 /srv/mongodb/rs0-1 /srv/mongodb/rs0-2
   ```

   (b) Issue the following commands, each in a distinct screen window:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/rs0-0 --replSet rs0
   mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0
   mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0
   ```

   This starts each instance as a member of a replica set named `rs0`, each running on a distinct port. If you are already using these ports, you can select different ports. See the documentation of the following options for more information: *--port* (page 582), *--dbpath* (page 583), and *--replSet* (page 586).

3. Open a `mongo` shell and connect to the first `mongod` instance. If you're running this command remotely, replace "localhost" with the appropriate hostname. In a new shell session, enter the following:

   ```
   mongo localhost:27017
   ```

4. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

5. Display the current *replica configuration* (page 661):

   ```
   rs.conf()
   ```

6. Add two members to the replica set by issuing a sequence of commands similar to the following.

   ```
   rs.add("localhost:27018")
   rs.add("localhost:27019")
   ```

   After these commands return you have a fully functional replica set. New replica sets elect a *primary* within a seconds.

7. 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 as an example of a basic automatically configured replica set.

[1] GNU Screen is packaged as `screen` on Debian-based, Fedora/Red Hat-based, and Arch Linux.

Deploy a Production Replica Set

Production replica sets are very similar to the development or testing deployment described above, with the following differences:

- Each member of the replica set resides on its own machine, and the MongoDB processes all bind to port `27017`, which is the standard MongoDB port.
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames in the following scheme:
  - `mongodb0.example.net`
  - `mongodb1.example.net`
  - `mongodb2.example.net`

  Configure DNS names appropriately, *or* set up your systems' `http://docs.mongodb.org/manual/etc/host` file to reflect this configuration.
- You specify run-time configuration on each system in a *configuration file* (page 621) stored in `http://docs.mongodb.org/manual/etc/mongodb.conf` or in a related location. You *do not* specify run-time configuration through command line options.

  For each MongoDB instance, use the following configuration. Set configuration values appropriate to your systems:

  ```
  port = 27017

  bind_ip = 10.8.0.10

  dbpath = /srv/mongodb/

  fork = true

  replSet = rs0
  ```

  You do not need to specify an interface with `bind_ip` (page 622). However, if you do not specify an interface, MongoDB listens for connections on all available IPv4 interfaces. Modify `bind_ip` (page 622) to reflect a secure interface on your system that is able to access all other members of the set *and* on which all other members of the replica set can access the current member. The DNS or host names must point and resolve to this IP address. Configure network rules or a virtual private network (i.e. "VPN") to permit this access.

  For more documentation on run time options used above and on additional configuration options, see *Configuration File Options* (page 621).

To deploy a production replica set:

1. Before creating your replica set, verify that every member can successfully connect to every other member. The network configuration must allow all possible connections between any any two members. To test connectivity, see *Test Connections Between all Members* (page 48).
2. On each system start the `mongod` process by issuing a command similar to following:

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

   Note: In production deployments you likely want to use and configure a *control script* to manage this process based on this command. Control scripts are beyond the scope of this document.

   ---

3. Open a `mongo` shell connected to this host:

```
mongo
```

4. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

5. Display the current *replica configuration* (page 661):

   ```
   rs.conf()
   ```

6. Add two members to the replica set by issuing a sequence of commands similar to the following.

   ```
   rs.add("mongodb1.example.net")
   rs.add("mongodb2.example.net")
   ```

   After these commands return you have a fully functional replica set. New replica sets elect a *primary* within a seconds.

7. 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()`

6.2 Convert a Standalone to a Replica Set

While *standalone* MongoDB instances are useful for testing, development and trivial deployments, for production use, *replica sets* provide required robustness and disaster recovery. This tutorial describes how to convert an existing standalone instance into a three-member replica set. If you're deploying a replica set "fresh," without any existing MongoDB data or instance, see *Deploy a Replica Set* (page 73).

For more information on on *replica sets, their use, and administration* (page 31), see:

- *Replication Fundamentals* (page 33),
- *Replication Architectures* (page 51),
- *Replica Set Administration* (page 38), and
- *Application Development with Replica Sets* (page 54).

6.2.1 Procedure

This procedure assumes you have a *standalone* instance of MongoDB installed. If you have not already installed MongoDB, see the *installation tutorials* (page 9).

1. Shut down the your MongoDB instance and then restart using the *--replSet* (page 586) option and the name of the *replica set*, which is `rs0` in the example below.

   Use a command similar to the following:

```
mongod --port 27017 --replSet rs0
```

This starts the instance as a member of a replica set named `rs0`. For more information on configuration options, see *Configuration File Options* (page 621) and the *mongod* (page 581).

2. Open a `mongo` shell and connect to the `mongod` instance. In a new system shell session, use the following command to start a `mongo` shell:

   ```
   mongo
   ```

3. Use `rs.initiate()` to initiate the replica set:

   ```
   rs.initiate()
   ```

   The set is now operational. To return the replica set configuration, call the `rs.conf()` method. To check the status of the replica set, use `rs.status()`.

4. Now add additional replica set members. On two distinct systems, start two new standalone `mongod` instances. Then, in the `mongo` shell instance connected to the first `mongod` instance, issue a command in the following form:

   ```
   rs.add("<hostname>:<port>")
   ```

   Replace `<hostname>` and `<port>` with the resolvable hostname and port of the `mongod` instance you want to add to the set. Repeat this operation for each `mongod` that you want to add to the set.

   For more information on adding hosts to a replica set, see the *Add Members to a Replica Set* (page 77) document.

6.3 Add Members to a Replica Set

6.3.1 Overview

This tutorial explains how to add an additional member to an existing replica set.

Before adding a new member, see the *Adding Members* (page 42) topic in the *Replica Set Administration* (page 38) document.

For background on replication deployment patterns, see the *Replication Architectures* (page 51) document.

6.3.2 Requirements

1. An active replica set.
2. A new MongoDB system capable of supporting your dataset, accessible by the active replica set through the network.

If neither of these conditions are satisfied, please use the MongoDB *installation tutorial* (page 9) and the *Deploy a Replica Set* (page 73) tutorial instead.

6.3.3 Procedures

The examples in this procedure use the following configuration:

- The active replica set is `rs0`.
- The new member to be added is `mongodb3.example.net`.

- The `mongod` instance default port is `27017`.
- The `mongodb.conf` configuration file exists in the `http://docs.mongodb.org/manual/etc` directory and contains the following replica set information:

```
port = 27017

bind_ip = 10.8.0.10

dbpath = /srv/mongodb/

logpath = /var/log/mongodb.log

fork = true

replSet = rs0
```

For more information on configuration options, see *Configuration File Options* (page 621).

Add a Member to an Existing Replica Set

This procedure uses the above *example configuration* (page 77).

1. Deploy a new `mongod` instance, specifying the name of the replica set. You can do this one of two ways:
   - Using the `mongodb.conf` file. On the *primary*, issue a command that resembles the following:

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

   - Using command line arguments. On the *primary*, issue command that resembles the following:

     ```
     mongod --replSet rs0
     ```

   Take note of the host name and port information for the new `mongod` instance.
2. Open a `mongo` shell connected to the replica set's primary:

   ```
   mongo
   ```

   Note: The primary is the only member that can add or remove members from the replica set. If you do not know which member is the primary, log into any member of the replica set using `mongo` and issue the `db.isMaster()` command to determine which member is in the `isMaster.primary` field. For example:

   ```
   mongo mongodb0.example.net

   db.isMaster()
   ```

   If you are not connected to the primary, disconnect from the current client and reconnect to the primary.

   ---
3. In the `mongo` shell, issue the following command to add the new member to the replica set.

   ```
   rs.add("mongodb3.example.net")
   ```

   Note: You can also 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 by calling the `rs.conf()` method, which displays the *replica set configuration* (page 661):

   ```
   rs.conf()
   ```

   You can use the `rs.status()` function to provide an overview of *replica set status* (page 659).

Add a Member to an Existing Replica Set (Alternate Procedure)

Alternately, you can add a member to a replica set by specifying an entire configuration document with some or all of the fields in a `members` (page 662) document. For example:

```
rs.add({_id: 1, host: "mongodb3.example.net:27017", priority: 0, hidden: true})
```

This configures a *hidden member* that is accessible at `mongodb3.example.net:27017`. See `host` (page 662), `priority` (page 663), and `hidden` (page 663) for more information about these settings. When you specify a full configuration object with `rs.add()`, you must declare the `_id` field, which is not automatically populated in this case.

6.3.4 Production Notes

- In production deployments you likely want to use and configure a *control script* to manage this process based on this command.
- A member can be removed from a set and re-added later. If the removed member's data is still relatively fresh, it can recover and catch up from its old data set. See the `rs.add()` and `rs.remove()` helpers.
- If you have a backup or snapshot of an existing member, you can move the data files (i.e. `http://docs.mongodb.org/manual/data/db` or `dbpath` (page 624)) to a new system and use them to quickly initiate a new member. These files must be:
  - clean: the existing dataset must be from a consistent copy of the database from a member of the same replica set. See the *Backup and Restoration Strategies* (page 180) document for more information.
  - recent: the copy must more recent than the oldest operation in the *primary* member's *oplog*. The new secondary must be able to become current using operations from the primary's oplog.
- There is a maximum of seven *voting members* (page 63) in any replica set. When adding more members to a replica set that already has seven votes, you must either:
  - add the new member as a *non-voting members* (page 42) or,
  - remove votes from an `existing member` (page 664).

6.4 Deploy a Geographically Distributed Replica Set

This tutorial describes how to deploy a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

See Also:

For appropriate background, see *Replication Fundamentals* (page 33) and *Replication Architectures* (page 51). For related tutorials, see *Deploy a Replica Set* (page 73) and *Add Members to a Replica Set* (page 77).

6.4.1 Overview

While *replica sets* provide basic protection against single-instance failure, when all of the members of a replica set reside within a single facility, the replica set is still susceptible to some classes of errors within that facility including power outages, networking distortions, and natural disasters. To protect against these classes of failures, deploy a replica set with one or more members in a geographically distinct facility or data center.

6.4.2 Requirements

For a three-member replica set you need two instances in a primary facility (hereafter, "Site A") and one member in a secondary facility (hereafter, "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.)

For a four-member replica set you need two members in Site A, two members in Site B (or one member in Site B and one member in Site C,) and a single *arbiter* in Site A.

For replica sets with additional members in the secondary facility or with multiple secondary facilities, the requirements are the same as above but with the following notes:

- Ensure that a majority of the *voting members* (page 42) are within Site A. This includes *secondary-only members* (page 39) and *arbiters* (page 41) For more information on the need to keep the voting majority on one site, see :ref'replica-set-elections-and-network-partitions'.
- If you deploy a replica set with an uneven number of members, deploy an *arbiter* (page 41) on Site A. The arbiter must be on site A to keep the majority there.

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.

6.4.3 Procedures

Deploy a Distributed Three-Member Replica Set

A geographically distributed three-member deployment has the following features:

- Each member of the replica set resides on its own machine, and the MongoDB processes all bind to port `27017`, which is the standard MongoDB port.
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames in the following scheme:
  - `mongodb0.example.net`
  - `mongodb1.example.net`
  - `mongodb2.example.net`

  Configure DNS names appropriately, *or* set up your systems' `http://docs.mongodb.org/manual/etc/host` file to reflect this configuration. Ensure that one system (e.g. `mongodb2.example.net`) resides in Site B. Host all other systems in Site A.
- Ensure that network traffic can pass between all members in the network securely and efficiently. Consider the following:
  - Establish a virtual private network between the systems in Site A and Site B to encrypt all traffic between the sites and remains private. Ensure that your network topology routs all traffic between members within a single site over the local area network.

- Configure authentication using `auth` (page 624) and `keyFile` (page 623), so that only servers and process 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.

  See Also:

  For more information on security and firewalls, see *Security Considerations for Replica Sets* (page 46).

- Specify run-time configuration on each system in a *configuration file* (page 621) stored in `http://docs.mongodb.org/manual/etc/mongodb.conf` or in a related location. *Do not* specify run-time configuration through command line options.

  For each MongoDB instance, use the following configuration, with values set appropriate to your systems:

  ```
  port = 27017

  bind_ip = 10.8.0.10

  dbpath = /srv/mongodb/

  fork = true

  replSet = rs0/mongodb0.example.net,mongodb1.example.net,mongodb2.example.net
  ```

  Modify `bind_ip` (page 622) to reflect a secure interface on your system that is able to access all other members of the set *and* that is accessible to all other members of the replica set. The DNS or host names need to point and resolve to this IP address. Configure network rules or a virtual private network (i.e. "VPN") to permit this access.

  Note: The portion of the `replSet` (page 628) following the `http://docs.mongodb.org/manual/` provides a "seed list" of known members of the replica set. `mongod` uses this list to fetch configuration changes following restarts. It is acceptable to omit this section entirely, and have the `replSet` (page 628) option resemble:

  ```
  replSet = rs0
  ```

  For more documentation on the above run time configurations, as well as additional configuration options, see *Configuration File Options* (page 621).

To deploy a geographically distributed three-member set:

1. On each system start the `mongod` process by issuing a command similar to following:

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

   Note: In production deployments you likely want to use and configure a *control script* to manage this process based on this command. Control scripts are beyond the scope of this document.

   ---

2. Open a `mongo` shell connected to this host:

   ```
   mongo
   ```

3. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

4. Display the current *replica configuration* (page 661):

   ```
   rs.conf()
   ```

5. Add the remaining members to the replica set by issuing a sequence of commands similar to the following. The example commands assume the current *primary* is `mongodb0.example.net`:

   ```
   rs.add("mongodb1.example.net")
   rs.add("mongodb2.example.net")
   ```

6. Make sure that you have configured the member located in Site B (i.e. `mongodb2.example.net`) as a *secondary-only member* (page 39):

   (a) Issue the following command to determine the `members[n]._id` (page 662) value for `mongodb2.example.net`:

   ```
   rs.conf()
   ```

   (b) In the `member array` (page 662), save the `members[n]._id` (page 662) value. The example in the next step assumes this value is `2`.

   (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)
   ```

   Note: In some situations, the `rs.reconfig()` shell command can force the current primary to step down and causes an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While, this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods.

   After these commands return you have a geographically distributed three-member replica set.

7. To check the status of your replica set, issue `rs.status()`.

See Also:

The documentation of the following shell functions for more information:

- `rs.initiate()`
- `rs.conf()`
- `rs.reconfig()`
- `rs.add()`

Deploy a Distributed Four-Member Replica Set

A geographically distributed four-member deployment has the following features:

- Each member of the replica set, except for the *arbiter* (see below), resides on its own machine, and the MongoDB processes all bind to port `27017`, which is the standard MongoDB port.
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames in the following scheme:
  - `mongodb0.example.net`

  - `mongodb1.example.net`
  - `mongodb2.example.net`
  - `mongodb3.example.net`

  Configure DNS names appropriately, *or* set up your systems' `http://docs.mongodb.org/manual/etc/host` file to reflect this configuration. Ensure that one system (e.g. `mongodb2.example.net`) resides in Site B. Host all other systems in Site A.

- One host (e.g. `mongodb3.example.net`) will be an *arbiter* and can run on a system that is also used for an application server or some other shared purpose.
- There are three possible architectures for this replica set:
  - Two members in Site A, two *secondary-only members* (page 39) in Site B, and an arbiter in Site A.
  - Three members in Site A and one secondary-only member in Site B.
  - Two members in Site A, one secondary-only member in Site B, one secondary-only member in Site C, and an arbiter in site A.

  In most cases the first architecture is preferable because it is the lest complex.
- Ensure that network traffic can pass between all members in the network securely and efficiently. Consider the following:
  - Establish a virtual private network between the systems in Site A and Site B (and Site C if it exists) to encrypt all traffic between the sites and remains private. Ensure that your network topology routs all traffic between members within a single site over the local area network.
  - Configure authentication using `auth` (page 624) and `keyFile` (page 623), so that only servers and process 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.

    See Also:

    For more information on security and firewalls, see *Security Considerations for Replica Sets* (page 46).
- Specify run-time configuration on each system in a *configuration file* (page 621) stored in `http://docs.mongodb.org/manual/etc/mongodb.conf` or in a related location. *Do not* specify run-time configuration through command line options.

  For each MongoDB instance, use the following configuration, with values set appropriate to your systems:

```
port = 27017

bind_ip = 10.8.0.10

dbpath = /srv/mongodb/

fork = true

replSet = rs0/mongodb0.example.net,mongodb1.example.net,mongodb2.example.net,mongodb3.example.ne
```

  Modify `bind_ip` (page 622) to reflect a secure interface on your system that is able to access all other members of the set *and* that is accessible to all other members of the replica set. The DNS or host names need to point and resolve to this IP address. Configure network rules or a virtual private network (i.e. "VPN") to permit this access.

Note: The portion of the `replSet` (page 628) following the `http://docs.mongodb.org/manual/` provides a "seed list" of known members of the replica set. `mongod` uses this list to fetch configuration changes following restarts. It is acceptable to omit this section entirely, and have the `replSet` (page 628) option resemble:

```
replSet = rs0
```

For more documentation on the above run time configurations, as well as additional configuration options, see doc:*/reference/configuration-options*.

To deploy a geographically distributed four-member set:

1. On each system start the `mongod` process by issuing a command similar to following:

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

   Note: In production deployments you likely want to use and configure a *control script* to manage this process based on this command. Control scripts are beyond the scope of this document.

   ---

2. Open a `mongo` shell connected to this host:

   ```
   mongo
   ```

3. Use `rs.initiate()` to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

4. Display the current *replica configuration* (page 661):

   ```
   rs.conf()
   ```

5. Add the remaining members to the replica set by issuing a sequence of commands similar to the following. The example commands assume the current *primary* is `mongodb0.example.net`:

   ```
   rs.add("mongodb1.example.net")
   rs.add("mongodb2.example.net")
   rs.add("mongodb3.example.net")
   ```

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 in Site B (e.g. `mongodb3.example.net`) as a *secondary-only member* (page 39):

   (a) Issue the following command to determine the `members[n]._id` (page 662) value for the member:

   ```
   rs.conf()
   ```

   (b) In the `member array` (page 662), save the `members[n]._id` (page 662) value. The example in the next step assumes this value is `2`.

   (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)
   ```

> **Note:** In some situations, the `rs.reconfig()` shell command can force the current primary to step down and causes an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While, this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods.

After these commands return you have a geographically distributed four-member replica set.

8. To check the status of your replica set, issue `rs.status()`.

See Also:

The documentation of the following shell functions for more information:

- `rs.initiate()`
- `rs.conf()`
- `rs.reconfig()`
- `rs.add()`

Deploy a Distributed Set with More than Four Members

The procedure for deploying a geographically distributed set with more than four members is similar to the above procedures, with the following differences:

- Never deploy more than seven voting members.
- Use the procedure for a four-member set if you have an even number of members (see *Deploy a Distributed Four-Member Replica Set* (page 82)). Ensure that Site A always has a majority of the members by deploying the *arbiter* within Site A. For six member sets, deploy at least three voting members in addition to the arbiter in Site A, the remaining members in alternate sites.
- Use the procedure for a three-member set if you have an odd number of members (see *Deploy a Distributed Three-Member Replica Set* (page 80)). Ensure that Site A always has a majority of the members of the set. For example, if a set has five members, deploy three remember within the primary facility and two remember 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.

6.5 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 36) 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 the *Oplog* (page 36) topic in the *Replication Fundamentals* (page 33) document. For details on the how oplog size affects *delayed members* and affects *replication lag*, see the *Delayed Members* (page 40) topic and the *Check the Replication Lag* (page 47) topic in *Replica Set Administration* (page 38).

6.5.1 Overview

The following is an overview of the procedure for changing the size of the oplog:

1. Shut down the current *primary* instance in the *replica set* and then restart it on a different port and in "standalone" mode.
2. Create a backup of the old (current) oplog. This is optional.
3. Save the last entry from the old oplog.
4. Drop the old oplog.
5. Create a new oplog of a different size.
6. Insert the previously saved last entry from the old oplog into the new oplog.
7. Restart the server as a member of the replica set on its usual port.
8. Apply this procedure to any other member of the replica set that *could become* primary.

6.5.2 Procedure

The examples in this procedure use the following configuration:

- The active *replica set* is `rs0`.
- The replica set is running on port is `27017`.
- The replica set is running with a `data directory` (page 624) of `http://docs.mongodb.org/manual/srv/mongodb`.

To change the size of the oplog for a replica set, use the following procedure for every member of the set that may become primary.

1. Shut down the `mongod` instance and restart it in "standalone" mode running on a different port.

   Note: Shutting down the *primary* member of the set will trigger a failover situation and another member in the replica set will become primary. In most cases, it is least disruptive to modify the oplogs of all the secondaries before modifying the primary.

   To shut down the current primary instance, use a command that resembles the following:

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

   To restart the instance on a different port and in "standalone" mode (i.e. without `replSet` (page 628) or *`--replSet`*), use a command that resembles the following:

   ```
   mongod --port 37017 --dbpath /srv/mongodb
   ```

2. Backup the existing oplog on the standalone instance. Use the following sequence of commands:

   ```
   mongodump --db local --collection 'oplog.rs' --port 37017
   ```

   Connect to the instance using the `mongo` shell:

   ```
   mongo --port 37017
   ```

3. Save the last entry from the old (current) oplog.

   (a) In the `mongo` shell, enter the following command to use the `local` database to interact with the oplog:

```
use local
```

(b) Use the `db.collection.save()` operation to save the last entry in the oplog to a temporary collection:

```
db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural : -1} ).limit(1).next
```

You can see this oplog entry in the `temp` collection by issuing the following command:

```
db.temp.find()
```

4. Drop the old `oplog.rs` collection in the `local` database. Use the following command:

```
db.oplog.rs.drop()

This will return ``true`` on the shell.
```

5. Use the `create` command to create a new oplog of a different size. Specify the `size` argument in bytes. A value of `2147483648` will create a new oplog that's 2 gigabytes:

```
db.runCommand( { create : "oplog.rs", capped : true, size : 2147483648 } )
```

Upon success, this command returns the following status:

```
{ "ok" : 1 }
```

6. Insert the previously saved last entry from the old oplog into the new oplog:

```
db.oplog.rs.save( db.temp.findOne() )
```

To confirm the entry is in the new oplog, issue the following command:

```
db.oplog.rs.find()
```

7. Restart the server as a member of the replica set on its usual port:

```
mongod --dbpath /srv/mongodb --shutdown
mongod --replSet rs0 --dbpath /srv/mongodb
```

The replica member will recover and "catch up" and then will be eligible for election to *primary*. To step down the "temporary" primary that took over when you initially shut down the server, use the `rs.stepDown()` method. This will force an election for primary. If the server's *priority* (page 34) is higher than all other members in the set *and* if it has successfully "caught up," then it will likely become primary.

8. Repeat this procedure for all other members of the replica set that are or could become primary.

6.6 Force a Member to Become Primary

6.6.1 Synopsis

You can force a *replica set* member to become *primary* by giving it a higher `members[n].priority` (page 663) value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `members[n].priority` (page 663) value to `0`, which means the member can never seek *election* (page 34) as primary. For more information, see *Secondary-Only Members* (page 39).

6.6.2 Procedures

Force a Member to be Primary by Setting its Priority High

Changed in version 2.0. For more information on priorities, see *Member Priority* (page 34).

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 665).

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 `members[n].priority` (page 663) value than the other `mongod` instances.

   The following sequence of events occur:

   - `m3.example.net` and `m2.example.net` sync with `m1.example.net` (typically within 10 seconds).
   - `m1.example.net` sees that it no longer has highest priority and, in most cases, steps down. `m1.example.net` *does not* step down if `m3.example.net`'s sync is far behind. In that case, `m1.example.net` waits until `m3.example.net` is within 10 seconds of its optime and then steps down. This minimizes the amount of time with no primary following failover.
   - The step down forces on election in which `m3.example.net` becomes primary based on its `priority` (page 663) 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` (page 519) commands.

6.7 Change Hostnames in a Replica Set

6.7.1 Synopsis

For most *replica sets* the hostnames [2] in the `members[n].host` (page 662) field never change. However, in some cases you must migrate some or all host names in a replica set as organizational needs change. This document presents two possible procedures for changing the hostnames in the `members[n].host` (page 662) field. Depending on your environments availability requirements, you may:

[2] Always use resolvable hostnames for the value of the `members[n].host` (page 662) field in the replica set configuration to avoid confusion and complexity.

1. Make the configuration change without disrupting the availability of the replica set. While this ensures that your application will always be able to read and write data to the replica set, this procedure can take a long time and may incur downtime at the application layer. [3]

   For this procedure, see *Changing Hostnames while Maintaining the Replica Set's Availability* (page 91).

2. Stop all members of the replica set at once running on the "old" hostnames or interfaces, make the configuration changes, and then start the members at the new hostnames or interfaces. While the set will be totally unavailable during the operation, the total maintenance window is often shorter.

   For this procedure, see *Changing All Hostnames in Replica Set at Once* (page 92).

See Also:

- *Replica Set Configuration* (page 661)
- *Replica Set Reconfiguration Process* (page 665)
- `rs.conf()` and `rs.reconfig()`

And the following tutorials:

- *Deploy a Replica Set* (page 73)
- *Add Members to a Replica Set* (page 77)

6.7.2 Procedures

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)

[3] You will have to configure your applications so that they can connect to the replica set at both the old and new locations. This often requires a restart and reconfiguration at the application layer, which may affect the availability of your applications. This re-configuration is beyond the scope of this document and makes the *second option* (page 92) preferable when you must change the hostnames of *all* members of the replica set at once.

- mongodb1.example.net:27017
- mongodb2.example.net:27017

Use the most appropriate procedure for your deployment.

Changing Hostnames while Maintaining the Replica Set's Availability

This procedure uses the above *assumptions* (page 90).

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) Run the following reconfigure option, for the members[n].host (page 662) value where n is 1:

   ```
   cfg = rs.conf()

   cfg.members[1].host = "mongodb1.example.net:27017"

   rs.reconfig(cfg)
   ```

   See *Replica Set Configuration* (page 661) for more information.
   (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 replSetStepDown (page 519). In the mongo shell, use the rs.stepDown() wrapper, as follows:

   ```
   rs.stepDown()
   ```

3. When the step down succeeds, shut down the primary.
4. To make the final configuration change, connect to the new primary in the mongo shell and reconfigure the members[n].host (page 662) value where n is 0:

   ```
   cfg = rs.conf()

   cfg.members[0].host = "mongodb0.example.net:27017"

   rs.reconfig(cfg)
   ```

5. Start the original primary.
6. Open a mongo shell connected to the primary.
7. To confirm the new configuration, call rs.conf() in the mongo shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
   ```

```
    "members" : [
        {
            "_id" : 0,
            "host" : "mongodb0.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "mongodb1.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017"
        }
    ]
}
```

Changing All Hostnames in Replica Set at Once

This procedure uses the above *assumptions* (page 90).

1. Stop all members in the *replica set*.
2. Restart each member *on a different port* and *without* using the *--replSet* (page 586) 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* (page 583), which in this example is `http://docs.mongodb.org/manual/data/db1`. Use a command that resembles the following:

   ```
   mongod --dbpath /data/db1/ --port 37017
   ```

3. For each member of the replica set, perform the following sequence of operations:

   (a) Open a `mongo` shell connected to the `mongod` running on the new, temporary port. For example, for a member running on a temporary port of `37017`, you would issue this command:

   ```
   mongo --port 37017
   ```

   (b) Edit the replica set configuration manually. The replica set configuration is the only document in the `system.replset` collection in the `local` database. Edit the replica set configuration with the new hostnames and correct ports for all the members of the replica set. Consider the following sequence of commands to change the hostnames in a three-member set:

   ```
   use local

   cfg = db.system.replset.findOne( { "_id": "rs" } )

   cfg.members[0].host = "mongodb0.example.net:27017"

   cfg.members[1].host = "mongodb1.example.net:27017"

   cfg.members[2].host = "mongodb2.example.net:27017"

   db.system.replset.update( { "_id": "rs" } , cfg )
   ```

   (c) Stop the `mongod` process on the member.

4. After re-configuring all members of the set, start each `mongod` instance in the normal way: use the usual port number and use the *--replSet* (page 586) option. For example:

```
mongod --dbpath /data/db1/ --port 27017 --replSet rs
```

5. Connect to one of the `mongod` instances using the `mongo` shell. For example:

   ```
   mongo --port 27017
   ```

6. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
           }
       ]
   }
   ```

6.8 Convert a Secondary to an Arbiter

If you have a *secondary* in a *replica set* that no longer needs to hold a copy of the data *but* that you want to retain in the set to ensure that the replica set will be able to *elect a primary* (page 34), you can convert the secondary into an *arbiter* (page 41). This document provides two equivalent procedures for this process.

6.8.1 Synopsis

Both of the following procedures are operationally equivalent. Choose whichever procedure you are most comfortable with:

1. 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 a Secondary to an Arbiter and Reuse the Port Number* (page 94).

2. 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 a Secondary to an Arbiter Running on a New Port Number* (page 94).

See Also:

- *Arbiters* (page 41)
- `rs.addArb()`
- *Replica Set Administration* (page 38)

6.8.2 Procedures

Convert a Secondary to an 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 a Secondary to an 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.

6.9 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 665).

This document provides the following options for reconfiguring a replica set when a **majority** of members are accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 95)
- *Reconfigure by Replacing the Replica Set* (page 96)

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 *Elections and Network Partitions* (page 64) for more information on this situation.

6.9.1 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 662) array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

   ```
   cfg.members = [cfg.members[0] , cfg.members[4] , cfg.members[7]]
   ```

4. On the same member, reconfigure the set by using the `rs.reconfig()` command with the `force` option set to `true`:

   ```
   rs.reconfig(cfg, {force : true})
   ```

   This operation forces the secondary to use the new configuration. The configuration is then propagated to all the surviving members listed in the `members` array. The replica set then elects a new primary.

   Note: When you use `force : true`, the version number in the replica set configuration increases significantly, by tens or hundreds of thousands. This is normal and designed to prevent set version collisions if you accidentally force reconfigs on both sides of a network partition and then the network partitioning ends.

   ---

5. If the failure or partition was only temporary, shut down or decommission the removed members as soon as possible.

6.9.2 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 95).

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 665).

If you run a pre-2.0 version and a majority of your replica set is down, you have the two options described here. Both involve replacing the replica set.

Reconfigure by Turning Off Replication

This option replaces the *replica set* with a *standalone* server.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control script* or an invocation that resembles the following:

   ```
   mongod --dbpath /data/db/ --shutdown
   ```

Set *--dbpath* (page 583) to the data directory of your mongod instance.

2. Create a backup of the data directory (i.e. dbpath (page 624)) of the surviving members of the set.

   Optional

   If you have a backup of the database you may instead remove this data.

3. Restart one of the mongod instances *without* the *--replSet* (page 586) 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. All other members will resync from this member.

1. Stop the surviving mongod instances. To ensure a clean shutdown, use an existing *control script* or an invocation that resembles the following:

   ```
   mongod --dbpath /data/db/ --shutdown
   ```

   Set *--dbpath* (page 583) to the data directory of your mongod instance.

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")
   ```

   These members will resync and copy all data from db0.example.net.

6.10 Recover MongoDB Data following Unexpected Shutdown

If MongoDB does not shutdown cleanly [4] the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption.

[4] To ensure a clean shut down, use the *mongod --shutdown* (page 586) option, your control script, "Control-C" (when running mongod in interactive mode,) or kill $(pidof mongod) or kill -2 $(pidof mongod).

To prevent data inconsistency and corruption, always shut down the database cleanly and use the *durability journaling* (page 625). The journal writes data to disk every 100 milliseconds by default and ensures that MongoDB can 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` (page 624) and allow MongoDB to resync the data.

See Also:

The *Administration* (page 161) documents, including *Syncing* (page 64), and the documentation on the `repair` (page 626), `repairpath` (page 627), and `journal` (page 625) settings.

6.10.1 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 *synchronize* (page 64) your data.

If the `mongod.lock` file in the data directory specified by `dbpath` (page 624), `http://docs.mongodb.org/manual/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 remove the lockfile and run repair. If you run repair when the `mongodb.lock` file exists without the *`mongod --repairpath`* (page 585) option, you will see a message that contains the following line:

```
old lock file: /data/db/mongod.lock. probably means unclean shutdown
```

You must remove the lockfile **and** run the repair operation before starting the database normally using 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 180) or resync from an intact member of the set, as described in *Resyncing a Member of a Replica Set* (page 45).

There are two processes to repair data files that result from an unexpected shutdown:

1. Use the *`--repair`* (page 585) option in conjunction with the *`--repairpath`* (page 585) 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`* (page 585) 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.

Procedures

To repair your data files using the *--repairpath* (page 585) option to preserve the original data files unmodified:

1. Start mongod using *--repair* (page 585) 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 http://docs.mongodb.org/manual/data/db0 directory.

2. Start mongod using the following invocation to point the dbpath (page 624) at http://docs.mongodb.org/manual/data/db2:

   ```
   mongod --dbpath /data/db0
   ```

   Once you confirm that the data files are operational you may delete or archive the data files in the http://docs.mongodb.org/manual/data/db directory.

To repair your data files without preserving the original files, do not use the *--repairpath* (page 585) option, as in the following procedure:

1. Remove the stale lock file:

   ```
   rm /data/db/mongod.lock
   ```

   Replace http://docs.mongodb.org/manual/data/db with your dbpath (page 624) where your MongoDB instance's data files reside.

   Warning: After you remove the mongod.lock file you *must* run the *--repair* (page 585) process before using your database.

2. Start mongod using *--repair* (page 585) 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 http://docs.mongodb.org/manual/data/db directory.

3. Start mongod using the following invocation to point the dbpath (page 624) at http://docs.mongodb.org/manual/data/db:

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

6.10.2 mongod.lock

In normal operation, you should **never** remove the mongod.lock file and start mongod. Instead use 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 180) or if running as a *replica set* resync from an intact member of the set, as described in *Resyncing a Member of a Replica Set* (page 45).

CHAPTER

SEVEN

REFERENCE

The following describes the replica set configuration object:

- *Replica Set Configuration* (page 661)

The following describe MongoDB output and status related to replication:

- *Replica Set Status Reference* (page 659)
- *Replication Info Reference* (page 667)

Part IV

Sharding

Sharding distributes a single logical database system across a cluster of machines. Sharding uses range-based portioning to distribute *documents* based on a specific *shard key*.

This page lists the documents, tutorials, and reference pages that describe sharding.

For an overview, see *Sharding Fundamentals* (page 107). To configure, maintain, and troubleshoot sharded clusters, see *Sharded Cluster Administration* (page 113). For deployment architectures, see *Sharded Cluster Architectures* (page 131). For details on the internal operations of sharding, see *Sharding Internals* (page 133). For procedures for performing certain sharding tasks, see the *Tutorials* (page 141) list.

CHAPTER

EIGHT

DOCUMENTATION

The following is the outline of the main documentation:

8.1 Sharding Fundamentals

This document provides an overview of the fundamental concepts and operations of sharding with MongoDB. For a list of all sharding documentation see *Sharding* (page 105).

MongoDB's sharding system allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` instances or *shards*. Sharding increases write capacity, provides the ability to support larger working sets, and raises the limits of total data size beyond the physical resources of a single node.

8.1.1 Sharding Overview

Features

With sharding MongoDB automatically distributes data among a collection of `mongod` instances. Sharding, as implemented in MongoDB has the following features:

Range-based Data Partitioning MongoDB distributes documents among *shards* based on the value of the *shard key* (page 109). Each *chunk* represents a block of *documents* with values that fall within a specific range. When chunks grow beyond the *chunk size* (page 137), MongoDB divides the chunks into smaller chunks (i.e. *splitting*) based on the shard key.

Automatic Data Volume Distribution The sharding system automatically balances data across the cluster without intervention from the application layer. Effective automatic sharding depends on a well chosen *shard key* (page 109), but requires no additional complexity, modifications, or intervention from developers.

Transparent Query Routing Sharding is completely transparent to the application layer, because all connections to a cluster go through `mongos` (page 676). Sharding in MongoDB requires some *basic initial configuration* (page 114), but ongoing function is entirely transparent to the application.

Horizontal Capacity Sharding increases capacity in two ways:

1. Effective partitioning of data can provide additional write capacity by distributing the write load over a number of `mongod` instances.
2. Given a shard key with sufficient *cardinality* (page 133), partitioning data allows users to increase the potential amount of data to manage with MongoDB and expand the *working set*.

A typical *sharded cluster* consists of:

- 3 config servers that store metadata. The metadata maps *chunks* to shards.

- More than one *replica sets* that hold data. These are the *shards*.
- A number of lightweight routing processes, called *mongos* (page 588) instances. The `mongos` (page 676) process routes operations to the correct shard based the cluster configuration.

When to Use Sharding

While sharding is a powerful and compelling feature, it comes with significant *Infrastructure Requirements* (page 108) and some limited complexity costs. As a result, use sharding only as necessary, and when indicated by actual operational requirements. Consider the following overview of indications it may be time to consider sharding.

You should consider deploying a *sharded cluster*, if:

- your data set approaches or exceeds the storage capacity of a single node in your system.
- the size of your system's active *working set will soon* exceed the capacity of the *maximum* amount of RAM for your system.
- your system has a large amount of write activity, a single MongoDB instance cannot write data fast enough to meet demand, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add additional complexity to your system without providing much benefit. When designing your data model, if you will eventually need a sharded cluster, consider which collections you will want to shard and the corresponding shard keys.

Warning: It takes time and resources to deploy sharding, and if your system has *already* reached or exceeded its capacity, you will have a difficult time deploying 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.

8.1.2 Sharding Requirements

Infrastructure Requirements

A *sharded cluster* has the following components:

- Three *config servers*.

  These special `mongod` instances store the metadata for the cluster. The `mongos` (page 676) instances cache this data and use it to determine which *shard* is responsible for which *chunk*.

  For development and testing purposes you may deploy a cluster with a single configuration server process, but always use exactly three config servers for redundancy and safety in production.

- Two or more shards. Each shard consists of one or more `mongod` instances that store the data for the shard.

  These "normal" `mongod` instances hold all of the actual data for the cluster.

  Typically each shard is a *replica sets*. Each replica set consists of multiple `mongod` instances. The members of the replica set provide redundancy and high available for the data in each shard.

  Warning: MongoDB enables data *partitioning*, or sharding, on a *per collection* basis. You *must* access all data in a sharded cluster via the `mongos` (page 676) instances as below. If you connect directly to a `mongod` in a sharded cluster you will see its fraction of cluster's data. The data on any given shard may be somewhat random: MongoDB provides no grantee that any two contiguous chunks will reside on a single shard.

- One or more mongos (page 676) instances.

  These instance direct queries from the application layer to the shards that hold the data. The mongos (page 676) instances have no persistent state or data files and only cache metadata in RAM from the config servers.

Note: In most situations mongos (page 676) instances use minimal resources, and you can run them on your application servers without impacting application performance. However, if you use the *aggregation framework* some processing may occur on the mongos (page 676) instances, causing that mongos (page 676) to require more system resources.

Data Requirements

Your cluster must manage a significant quantity of data for sharding to have an effect on your collection. The default *chunk* size is 64 megabytes, [1] and the *balancer* (page 112) will not begin moving data until the imbalance of chunks in the cluster exceeds the *migration threshold* (page 137).

Practically, this means that unless your cluster has many hundreds of megabytes of data, chunks will remain on a single shard.

While there are some exceptional situations where you may need to shard a small collection of data, most of the time the additional complexity added by sharding the small collection is not worth the additional complexity and overhead unless you need additional concurrency or capacity for some reason. If you have a small data set, usually a properly configured single MongoDB instance or replica set will be more than sufficient for your persistence layer needs.

Sharding and "localhost" Addresses

Because all components of a *sharded cluster* must communicate with each other over the network, there are special restrictions regarding the use of localhost addresses:

If you use either "localhost" or "127.0.0.1" as the host identifier, then you must use "localhost" or "127.0.0.1" for *all* host settings for any MongoDB instances in the cluster. This applies to both the host argument to addShard and the value to the *mongos --configdb* (page 590) run time option. If you mix localhost addresses with remote host address, MongoDB will produce errors.

8.1.3 Shard Keys

"Shard keys" refer to the *field* that exists in every *document* in a collection that MongoDB uses to distribute documents among the *shards*. Shard keys, like *indexes*, can be either a single field, or may be a compound key, consisting of multiple fields.

Remember, MongoDB's sharding is range-based: each *chunk* holds documents having specific range of values for the "shard key". Thus, 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 application queries and writes data to the database.

The ideal shard key:

- is easily divisible which makes it easy for MongoDB to distribute content among the shards. Shard keys that have a limited number of possible values are not ideal as they can result in some chunks that are "unsplitable." See the *Cardinality* (page 133) section for more information.

[1] *chunk* size is *user configurable* (page 590). However, the default value is of 64 megabytes is ideal for most deployments. See the *Chunk Size* (page 137) section in the *Sharding Internals* (page 133) document for more information.

- will distribute write operations among the cluster, to prevent any single shard from becoming a bottleneck. Shard keys that have a high correlation with insert time are poor choices for this reason; however, shard keys that have higher "randomness" satisfy this requirement better. See the *Write Scaling* (page 134) section for additional background.
- will make it possible for the `mongos` (page 676) to return most query operations directly from a single *specific* `mongod` instance. Your shard key should be the primary field used by your queries, and fields with a high degree of "randomness" are poor choices for this reason. See the *Query Isolation* (page 134) section for specific examples.

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.

8.1.4 Config Servers

Config servers maintain the shard metadata in a config database. The *config database* stores the relationship between *chunks* and where they reside within a *sharded cluster*. Without a config database, the `mongos` (page 676) instances would be unable to route queries or write operations within the cluster.

Config servers *do not* run as replica sets. Instead, a *cluster* operates with a group of *three* config servers that use a two-phase commit process that ensures immediate consistency and reliability.

For testing purposes you may deploy a cluster with a single config server, but this is not recommended for production.

> **Warning:** If your cluster has a single config server, this `mongod` is a single point of failure. If the instance 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.

The actual load on configuration servers is small because each `mongos` (page 676) instances maintains a cached copy of the configuration database. MongoDB only writes data to the config server to:

- create splits in existing chunks, which happens as data in existing chunks exceeds the maximum chunk size.
- migrate a chunk between shards.

Additionally, all config servers must be available on initial setup of a sharded cluster, each `mongos` (page 676) instance must be able to write to the `config.version` collection.

If one or two configuration instances become unavailable, the cluster's metadata becomes *read only*. It is still possible to read and write data from the shards, but no chunk migrations or splits will occur until all three servers are accessible. At the same time, config server data is only read in the following situations:

- A new `mongos` (page 676) starts for the first time, or an existing `mongos` (page 676) restarts.
- After a chunk migration, the `mongos` (page 676) instances update themselves with the new cluster metadata.

If all three config servers are inaccessible, you can continue to use the cluster as long as you don't restart the `mongos` (page 676) instances until the after config servers are accessible again. If you restart the `mongos` (page 676) instances and there are no accessible config servers, the `mongos` (page 676) would be unable to direct queries or write operations to the cluster.

Because the configuration data is small relative to the amount of data stored in a cluster, the amount of activity is relatively low, and 100% up time is not required for a functioning sharded cluster. As a result, backing up the config servers is not difficult. Backups of config servers are critical as clusters become totally inoperable when you lose all configuration instances and data. Precautions to ensure that the config servers remain available and intact are critical.

Note: Configuration servers store metadata for a single sharded cluster. You must have a separate configuration server or servers for each cluster you administer.

8.1.5 `mongos` and Querying

See Also:

mongos (page 588) and the `mongos` (page 676)-only settings: `test` (page 630) and `chunkSize` (page 630).

Operations

The `mongos` (page 676) provides a single unified interface to a sharded cluster for applications using MongoDB. Except for the selection of a *shard key*, application developers and administrators need not consider any of the *internal details of sharding* (page 133).

`mongos` (page 676) caches data from the *config server* (page 110), and uses this to route operations from applications and clients to the `mongod` instances. `mongos` (page 676) have no *persistent* state and consume minimal system resources.

The most common practice is to run `mongos` (page 676) instances on the same systems as your application servers, but you can maintain `mongos` (page 676) 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` (page 676) 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

`mongos` (page 676) uses information from *config servers* (page 110) to route operations to the cluster as efficiently as possible. In general, operations in a sharded environment are either:

1. Targeted at a single shard or a limited group of shards based on the shard key.
2. Broadcast to all shards in the cluster that hold documents in a collection.

When possible you should design your operations to be as targeted as possible. Operations have the following targeting characteristics:

- Query operations broadcast to all shards [2] **unless** the `mongos` (page 676) can determine which shard or shard stores this data.

  For queries that include the shard key, `mongos` (page 676) can target the query at a specific shard or set set of shards, 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` (page 676) *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 }
  ```

[2] If a shard does not store chunks from a given collection, queries for documents in that collection are not broadcast to that shard.

Depending on the distribution of data in the cluster and the selectivity of the query, mongos (page 676) may still have to contact multiple shards [3] to fulfill these queries.

- All `insert()` operations target to one shard.
- All single `update()` operations target to one shard. This includes *upsert* operations.
- The mongos (page 676) broadcasts multi-update operations to every shard.
- The mongos (page 676) broadcasts `remove()` operations to every shard unless the operation specifies the shard key in full.

While some operations must broadcast to all shards, you can improve performance by using as many targeted operations as possible by ensuring that your operations include the shard key.

Sharded Query Response Process

To route a query to a *cluster*, mongos (page 676) uses the following process:

1. Determine the list of *shards* that must receive the query.

   In some cases, when the *shard key* or a prefix of the shard key is a part of the query, the mongos (page 676) can route the query to a subset of the shards. Otherwise, the mongos (page 676) must direct the query to *all* shards.

   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 (page 676) 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 }
   ```

   ---

2. Establish a cursor on all targeted shards.

   When the first batch of results return from the cursors:

   (a) For query with sorted results (i.e. using `cursor.sort()`) the mongos (page 676) performs a merge sort of all queries.

   (b) For a query with unsorted results, the mongos (page 676) returns a result cursor that "round robins" results from all cursors on the shards. Changed in version 2.0.5: Before 2.0.5, the mongos (page 676) exhausted each cursor, one by one.

8.1.6 Balancing and Distribution

Balancing is the process MongoDB uses to redistribute data within a *sharded cluster*. When a *shard* has a too many: term:*chunks* <*chunk*> when compared to other shards, MongoDB balances the shards.

The balancing process attempts to minimize the impact that balancing can have on the cluster, by:

- Moving only one chunk at a time.

[3] mongos (page 676) will round some queries, even some that include the shard key, to all shards, if needed.

- Only initiating a balancing round when the difference in number of chunks between the shard with the greatest and the shard with the least number of chunks exceeds the *migration threshold* (page 137).

You may disable the balancer on a temporary basis for maintenance and limit the window during which it runs to prevent the balancing process from impacting production traffic.

See Also:

Balancer Operations (page 124) and *Cluster Balancer* (page 136).

Note: The balancing procedure for *sharded clusters* is entirely transparent to the user and application layer. This documentation is only included for your edification and possible troubleshooting purposes.

8.1.7 Security Considerations for Sharded Clusters

Note: You should always run all `mongod` components in trusted networking environments that control access to the cluster using network rules and restrictions to ensure that only known traffic reaches your `mongod` and `mongos` (page 676) instances.

Warning: Limitations Changed in version 2.2: Read only authentication is fully supported in shard clusters. Previously, in version 2.0, sharded clusters would not enforce read-only limitations.Changed in version 2.0: Sharded clusters support authentication. Previously, in version 1.8, sharded clusters will not support authentication and access control. You must run your sharded systems in trusted environments.

To control access to a sharded cluster, you must set the `keyFile` (page 623) option on all components of the sharded cluster. Use the `--keyFile` (page 590) run-time option or the `keyFile` (page 623) configuration option for all `mongos` (page 676), configuration instances, and shard `mongod` instances.

There are two classes of security credentials in a sharded cluster: credentials for "admin" users (i.e. for the *admin database*) and credentials for all other databases. These credentials reside in different locations within the cluster and have different roles:

1. Admin database credentials reside on the config servers, to receive admin access to the cluster you *must* authenticate a session while connected to a `mongos` (page 676) instance using the *admin database*.
2. Other database credentials reside on the *primary* shard for the database.

This means that you *can* authenticate to these users and databases while connected directly to the primary shard for a database. However, for clarity and consistency all interactions between the client and the database should use a `mongos` (page 676) instance.

Note: Individual shards can store administrative credentials to their instance, which only permit access to a single shard. MongoDB stores these credentials in the shards' *admin databases* and these credentials are *completely* distinct from the cluster-wide administrative credentials.

8.2 Sharded Cluster Administration

This document describes common administrative tasks for sharded clusters. For complete documentation of sharded clusters see the *Sharding* (page 105) section of this manual.

Sharding Procedures:

8.2.1 Set up a Sharded Cluster

Before deploying a cluster, see *Sharding Requirements* (page 108).

For testing purposes, you can run all the required shard `mongod` processes on a single server. For production, use the configurations described in *Replication Architectures* (page 51).

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 *`mongos --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.

If you have an existing replica set, you can use the *Convert a Replica Set to a Replicated Sharded Cluster* (page 195) tutorial as a guide. If you're deploying a cluster from scratch, see the *Deploy a Sharded Cluster* (page 141) tutorial for more detail or use the following procedure as a quick starting point:

1. Create data directories for each of the three (3) config server instances.

2. Start the three config server instances. For example, to start a config server instance running on TCP port `27018` with the data stored in `http://docs.mongodb.org/manual/data/configdb`, type the following:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27018
   ```

   For additional command options, see *mongod* (page 581) and *Configuration File Options* (page 621).

   Note: All config servers must be running and available when you first initiate a *sharded cluster*.

   ---

3. Start a `mongos` (page 676) instance. For example, to start a `mongos` (page 676) that connects to config server instance running on the following hosts:

   - `mongoc0.example.net`
   - `mongoc1.example.net`
   - `mongoc2.example.net`

   You would issue the following command:

   ```
   mongos --configdb mongoc0.example.net:27018,mongoc1.example.net:27018,mongoc2.example.net:27018
   ```

4. Connect to one of the `mongos` (page 676) instances. For example, if a `mongos` (page 676) is accessible at `mongos0.example.net` on port `27017`, issue the following command:

   ```
   mongo mongos0.example.net
   ```

5. Add shards to the cluster.

   Note: In production deployments, all shards should be replica sets.

   To deploy a replica set, see the *Deploy a Replica Set* (page 73) tutorial.

   From the `mongo` shell connected to the `mongos` (page 676) instance, call the `sh.addShard()` method for each shard to add to the cluster.

   For example:

   ```
   sh.addShard( "mongodb0.example.net:27027" )
   ```

   If `mongodb0.example.net:27027` is a member of a replica set, call the `sh.addShard()` method with an argument that resembles the following:

   ```
   sh.addShard( "<setName>/mongodb0.example.net:27027" )
   ```

   Replace, `<setName>` with the name of the replica set, and MongoDB will discover all other members of the replica set. Repeat this step for each new shard in your cluster.

   Optional

   You can specify a name for the shard and a maximum size. See `addShard`.

   Note: Changed in version 2.0.3. Before version 2.0.3, you must specify the shard in the following form:

   ```
   replicaSetName/<seed1>,<seed2>,<seed3>
   ```

   For example, if the name of the replica set is `repl0`, then your `sh.addShard()` command would be:

```
sh.addShard( "repl0/mongodb0.example.net:27027,mongodb1.example.net:27017,mongodb2.example.net:2
```

6. Enable sharding for each database you want to shard. While sharding operates on a per-collection basis, you must enable sharding for each database that holds collections you want to shard. This step is a meta-data change and will not redistribute your data.

   MongoDB enables sharding on a per-database basis. This is only a meta-data change and will not redistribute your data. To enable sharding for a given database, use the `enableSharding` command or the `sh.enableSharding()` shell helper.

   ```
   db.runCommand( { enableSharding: <database> } )
   ```

   Or:

   ```
   sh.enableSharding(<database>)
   ```

   Note: MongoDB creates databases automatically upon their first use.

   Once you enable sharding for a database, MongoDB assigns a *primary shard* for that database, where MongoDB stores all data before sharding begins.

1. Enable sharding on a per-collection basis.

   Finally, you must explicitly specify collections to shard. The collections must belong to a database for which you have enabled sharding. When you shard a collection, you also choose the shard key. To shard a collection, run the `shardCollection` command or the `sh.shardCollection()` shell helper.

   ```
   db.runCommand( { shardCollection: "<database>.<collection>", key: "<shard-key>" } )
   ```

   Or:

   ```
   sh.shardCollection("<database>.<collection>", "key")
   ```

   For example:

   ```
   db.runCommand( { shardCollection: "myapp.users", key: {username: 1} } )
   ```

   Or:

   ```
   sh.shardCollection("myapp.users", {username: 1})
   ```

   The choice of shard key is incredibly important: it affects everything about the cluster from the efficiency of your queries to the distribution of data. Furthermore, you cannot change a collection's shard key after setting it.

   See the *Shard Key Overview* (page 109) and the more in depth documentation of *Shard Key Qualities* (page 133) to help you select better shard keys.

   If you do not specify a shard key, MongoDB will shard the collection using the `_id` field.

8.2.2 Cluster Management

This section outlines procedures for adding and remove shards, as well as general monitoring and maintenance of a *sharded cluster*.

Add a Shard to a Cluster

To add a shard to an *existing* sharded cluster, use the following procedure:

1. Connect to a `mongos` (page 676) in the cluster using the `mongo` shell.
2. First, you need to tell the cluster where to find the individual shards. You can do this using the `addShard` command or the `sh.addShard()` helper:

   ```
   sh.addShard( "<hostname>:<port>" )
   ```

   Replace `<hostname>` and `<port>` with the hostname and TCP port number of where the shard is accessible.

   For example:

   ```
   sh.addShard( "mongodb0.example.net:27027" )
   ```

   Note: In production deployments, all shards should be replica sets.

   Repeat for each shard in your cluster.

   Optional

   You may specify a "name" as an argument to the `addShard`, as follows:

   ```
   db.runCommand( { addShard: mongodb0.example.net, name: "mongodb0" } )
   ```

   You cannot specify a name for a shard using the `sh.addShard()` helper in the `mongo` shell. If you use the helper or do not specify a shard name, then MongoDB will assign a name upon creation.

   Changed in version 2.0.3: Before version 2.0.3, you must specify the shard in the following form: the replica set name, followed by a forward slash, followed by a comma-separated list of seeds for the replica set. For example, if the name of the replica set is "myapp1", then your `sh.addShard()` command might resemble:

   ```
   sh.addShard( "repl0/mongodb0.example.net:27027,mongodb1.example.net:27017,mongodb2.example.net:2
   ```

Note: It may take some time for *chunks* to migrate to the new shard.

For an introduction to balancing, see *Balancing and Distribution* (page 112). For lower level information on balancing, see *Cluster Balancer* (page 136).

Remove a Shard from a Cluster

To remove a *shard* from a *sharded cluster*, you must:

- Migrate *chunks* to another shard or database.
- Ensure that this shard is not the *primary shard* for any databases in the cluster. If it is, move the "primary" status for these databases to other shards.
- Finally, remove the shard from the cluster's configuration.

Note: To successfully migrate data from a shard, the *balancer* process **must** be active.

The procedure to remove a shard is as follows:

1. Connect to a `mongos` (page 676) in the cluster using the `mongo` shell.
2. Determine the name of the shard you will be removing.

   You must specify the name of the shard. You may have specified this shard name when you first ran the `addShard` command. If not, you can find out the name of the shard by running the `listShards` or `printShardingStatus` commands or the `sh.status()` shell helper.

   The following examples will remove a shard named `mongodb0` from the cluster.
3. Begin removing chunks from the shard.

   Start by running the `removeShard` command. This will start "draining" or migrating chunks from the shard you're removing to another shard in the cluster.

   ```
   db.runCommand( { removeshard: "mongodb0" } )
   ```

   This operation will return the following response immediately:

   ```
   { msg : "draining started successfully" , state: "started" , shard :"mongodb0" , ok : 1 }
   ```

   Depending on your network capacity and the amount of data in the shard, this operation can take anywhere from a few minutes to several days to complete.
4. View progress of the migration.

   You can run the `removeShard` again at any stage of the process to view the progress of the migration, as follows:

   ```
   db.runCommand( { removeShard: "mongodb0" } )
   ```

   The output should look something like this:

   ```
   { msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 42, dbs : 1 }, ok: 1 }
   ```

   In the `remaining` sub-document `{ chunks: xx, dbs: y }`, a counter 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 remaining number of chunks to transfer is 0.
5. Move any databases to other shards in the cluster as needed.

   This is only necessary when removing a shard that is also the *primary shard* for one or more databases.

   Issue the following command at the `mongo` shell:

   ```
   db.runCommand( { movePrimary: "myapp", to: "mongodb1" })
   ```

   This command will migrate all remaining non-sharded data in the database named `myapp` to the shard named `mongodb1`.

   > **Warning:** Do not run the `movePrimary` until you have *finished* draining the shard.

   The command will not return until MongoDB completes moving all data. The response from this command will resemble the following:

   ```
   { "primary" : "mongodb1", "ok" : 1 }
   ```
6. Run `removeShard` again to clean up all metadata information and finalize the shard removal, as follows:

```
db.runCommand( { removeshard: "mongodb0" } )
```

When successful, this command will return a document like this:

```
{ msg: "remove shard completed successfully" , stage: "completed", host: "mongodb0", ok : 1 }
```

Once the value of the `stage` field is "completed," you may safely stop the processes comprising the `mongodb0` shard.

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the *Config Database Contents* (page 675). A database has sharding enabled if the value of the `partitioned` field is `true`. Connect to a `mongos` (page 676) 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 the `animals` database.

List Shards

To list the current set of configured shards, use the `listShards` command, as follows:

```
db.runCommand( { list shards : 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" : "animals",  "partitioned" : true,  "primary" : "shard0000" }
        foo.big chunks:
                shard0001    1
                shard0000    6
            { "a" : { $minKey : 1 } } -->> { "a" : "elephant" } on : shard0001 Timestamp(2000, 1) jur
            { "a" : "elephant" } -->> { "a" : "giraffe" } on : shard0000 Timestamp(1000, 1) jumbo
            { "a" : "giraffe" } -->> { "a" : "hippopotamus" } on : shard0000 Timestamp(2000, 2) jumbo
            { "a" : "hippopotamus" } -->> { "a" : "lion" } on : shard0000 Timestamp(2000, 3) jumbo
            { "a" : "lion" } -->> { "a" : "rhinoceros" } on : shard0000 Timestamp(1000, 3) jumbo
            { "a" : "rhinoceros" } -->> { "a" : "springbok" } on : shard0000 Timestamp(1000, 4)
            { "a" : "springbok" } -->> { "a" : { $maxKey : 1 } } on : shard0000 Timestamp(1000, 5)
        foo.large chunks:
                shard0001    1
                shard0000    5
            { "a" : { $minKey : 1 } } -->> { "a" : "hen" } on : shard0001 Timestamp(2000, 0)
            { "a" : "hen" } -->> { "a" : "horse" } on : shard0000 Timestamp(1000, 1) jumbo
            { "a" : "horse" } -->> { "a" : "owl" } on : shard0000 Timestamp(1000, 2) jumbo
            { "a" : "owl" } -->> { "a" : "rooster" } on : shard0000 Timestamp(1000, 3) jumbo
            { "a" : "rooster" } -->> { "a" : "sheep" } on : shard0000 Timestamp(1000, 4)
            { "a" : "sheep" } -->> { "a" : { $maxKey : 1 } } on : shard0000 Timestamp(1000, 5)
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
```

8.2.3 Chunk Management

This section describes various operations on *chunks* in *sharded clusters*. MongoDB automates most chunk management operations. However, these chunk management operations are accessible to administrators for use in some situations, typically surrounding initial setup, deployment, and data ingestion.

Split Chunks

Normally, MongoDB splits a *chunk* following inserts when a chunk exceeds the *chunk size* (page 137). The *balancer* may migrate recently split chunks to a new shard immediately if `mongos` (page 676) predicts future insertions will benefit from the move.

The MongoDB treats all chunks the same, whether split manually or automatically by the system.

> **Warning:** You cannot merge or combine chunks once you have split them.

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.

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.

Use `sh.status()` to determine the current chunks ranges across the cluster.

To split chunks manually, use the `split` command with operators: `middle` and `find`. The equivalent shell helpers are `sh.splitAt()` or `sh.splitFind()`.

Example

The following command will split 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 } )
```

`sh.splitFind()` will split 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 `sh.splitFind()`. The query in `sh.splitFind()` need not contain the shard key, though it almost always makes sense to query for the shard key in this case, and including the shard key will expedite the operation.

Use `sh.splitAt()` to split a chunk in two using the queried document as the partition point:

```
sh.splitAt( "records.people", { "zipcode": 63109 } )
```

However, the location of the document that this query finds with respect to the other documents in the chunk does not affect how the chunk splits.

Create Chunks (Pre-Splitting)

In most situations a *sharded cluster* will create and distribute chunks automatically without user intervention. However, in a limited number of use profiles, MongoDB cannot create enough chunks or distribute data fast enough to support required throughput. Consider the following scenarios:

- you must partition an existing data collection that resides on a single shard.
- you must ingest a large volume of data into a cluster that isn't balanced, or where the ingestion of data will lead to an imbalance of data.

  This can arise in an initial data loading, or in a case where you must insert a large volume of data into a single chunk, as is the case when you must insert at the beginning or end of the chunk range, as is the case for monotonically increasing or decreasing shard keys.

Preemptively splitting chunks increases cluster throughput for these operations, by reducing the overhead of migrating chunks that hold data during the write operation. MongoDB only creates splits after an insert operation, and can only migrate a single chunk at a time. Chunk migrations are resource intensive and further complicated by large write volume to the migrating chunk.

To create and migrate chunks manually, use the following procedure:

1. Split empty chunks in your collection by manually performing `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.

2. Migrate chunks manually using the `moveChunk` command:

Example

To migrate all of the manually created user profiles evenly, putting each prefix chunk on the next shard from the other, run the following commands in the mongo shell:

```
var shServer = [ "sh0.example.net", "sh1.example.net", "sh2.example.net", "sh3.example.net",
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
  }
}
```

You can also let the balancer automatically distribute the new chunks. For an introduction to balancing, see *Balancing and Distribution* (page 112). For lower level information on balancing, see *Cluster Balancer* (page 136).

Modify Chunk Size

When you initialize a sharded cluster, the default chunk size is 64 megabytes. This default chunk size works well for most deployments. However, if you notice that automatic migrations are incurring a level of I/O that your hardware cannot handle, you may want to reduce the chunk size. For the 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` (page 676) in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database Contents* (page 675):

   ```
   use config
   ```

3. Issue the following `save()` operation:

   ```
   db.settings.save( { _id:"chunksize", value: <size> } )
   ```

   Where the value of `<size>` reflects the new chunk size in megabytes. Here, you're essentially writing a document whose values store the global chunk size configuration value.

Note: The `chunkSize` (page 630) and `--chunkSize` (page 590) options, passed at runtime to the `mongos` (page 676) **do not** affect the chunk size after you have initialized the cluster.

To eliminate confusion you should *always* set chunk size using the above procedure and never use the runtime options.

Modifying the chunk size has several limitations:

- Automatic splitting only occurs when inserting *documents* or updating existing documents.

- 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 insertion or updates until they reach the new size.

Migrate Chunks

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:

- If you create chunks by *pre-splitting* the data in your collection, you will have to migrate chunks manually to distribute chunks 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, then you will have to migrate chunks manually.

For more information on how chunks move between shards, see *Cluster Balancer* (page 136), in particular the section *Chunk Migration* (page 138).

To migrate chunks, use the `moveChunk` command.

Note: To return a list of shards, use the `listShards` command.

Specify shard names using the `addShard` command using the `name` argument. If you do not specify a name in the `addShard` command, MongoDB will assign a name automatically.

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* you want to migrate.

To move this chunk, you would issue the following command from a `mongo` shell connected to any `mongos` (page 676) instance.

```
db.adminCommand({moveChunk : "myapp.users", find : {username : "smith"}, to : "mongodb-shard3.example
```

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.

See *Create Chunks (Pre-Splitting)* (page 121) for an introduction to pre-splitting. New in version 2.2: `moveChunk` command has the: `_secondaryThrottle` parameter. When set to `true`, MongoDB ensures that *secondary* members have replicated operations before allowing new chunk migrations.

Warning: The `moveChunk` command may produce the following error message:

```
The collection's metadata lock is already taken.
```

These errors occur when clients have too many open *cursors* that access the chunk you are migrating. You can either wait until the cursors complete their operation or close the cursors manually.

Strategies for Bulk Inserts in Sharded Clusters

Large bulk insert operations including initial data ingestion or routine data import, can have a significant impact on a *sharded cluster*. Consider the following strategies and possibilities for bulk insert operations:

- If the collection does not have data, then there is only one *chunk*, which must reside on a single shard. MongoDB must receive data, create splits, and distribute chunks to the available shards. To avoid this performance cost, you can pre-split the collection, as described in *Create Chunks (Pre-Splitting)* (page 121).

- You can parallels import by sending insert operations to more than one `mongos` (page 676) instance. If the collection is empty, pre-split first, as described in *Create Chunks (Pre-Splitting)* (page 121).
- If your shard key increases monotonically during an insert then all the inserts will go 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 a single shard.

  If your insert volume is never larger than what a single shard can process, then there is no problem; however, if the insert volume exceeds that range, and you cannot avoid a monotonically increasing shard key, then consider the following modifications to your application:

  - Reverse all the bits of the shard key to preserves the information while avoiding the correlation of insertion order and 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 might insert o into a sharded collection...
}
```

For information on choosing a shard key, see *Shard Keys* (page 109) and see *Shard Key Internals* (page 133) (in particular, *Operations and Reliability* (page 135) and *Choosing a Shard Key* (page 135)).

8.2.4 Balancer Operations

This section describes provides common administrative procedures related to balancing. For an introduction to balancing, see *Balancing and Distribution* (page 112). For lower level information on balancing, see *Cluster Balancer* (page 136).

Check the Balancer Lock

To see if the balancer process is active in your *cluster*, do the following:

1. Connect to any `mongos` (page 676) in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database Contents* (page 675):

   ```
   use config
   ```

3. Use the following query to return the balancer lock:

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

You can also use the following shell helper to return the same information:

```
sh.getBalancerState().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" }
```

Here's what this tells you:

- The balancer originates from the `mongos` (page 676) running on the system with the hostname `mongos0.example.net`.
- The value in the `state` field indicates that a `mongos` (page 676) has the lock. For version 2.0 and later, the value of an active lock is `2`; for earlier versions the value is `1`.

Note: Use the `sh.isBalancerRunning()` helper in the `mongo` shell to determine if the balancer is running, as follows:

```
sh.isBalancerRunning()
```

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` (page 676) in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database Contents* (page 675):

   ```
   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>", sto
   ```

   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` (page 676) instance in the sharded cluster. For instance, running the following will force the balancer to run between 11PM and 6AM local time only:

   ```
   db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "23:00", stop : "6:
   ```

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 125) 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` (page 676) in the cluster using the `mongo` shell.
1. Issue *one* of the following operations to disable the balancer:

   ```
   sh.setBalancerState(false)
   sh.stopBalancer()
   ```

2. Later, issue *one* the following operations to enable the balancer:

   ```
   sh.setBalancerState(true)
   sh.startBalancer()
   ```

Note: If a balancing round is in progress, the system will complete the current round before the balancer is officially disabled. After disabling, you can use the `sh.getBalancerState()` shell function to determine whether the balancer is in fact disabled.

The above process and the `sh.setBalancerState()`, `sh.startBalancer()`, and `sh.stopBalancer()` helpers provide wrappers on the following process, which may be useful if you need to run this operation from a driver that does not have helper functions:

1. Connect to any `mongos` (page 676) in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database Contents* (page 675):

   ```
   use config
   ```

3. Issue the following update to disable the balancer:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
   ```

4. To enable the balancer again, alter the value of "stopped" as follows:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , true );
   ```

8.2.5 Config Server Maintenance

Config servers store all cluster metadata, most importantly, the mapping from *chunks* to *shards*. This section provides an overview of the basic procedures to migrate, replace, and maintain these servers.

See Also:

Config Servers (page 110)

Deploy Three Config Servers for Production Deployments

For redundancy, all production *sharded clusters* should deploy three config servers processes on three different machines.

Do not use only a single config server for production deployments. Only use a single config server deployments for testing. You should upgrade to three config servers immediately if you are shifting to production. The following process shows how to convert a test deployment with only one config server to production deployment with three config servers.

1. Shut down all existing MongoDB processes. This includes:
   - all `mongod` instances or *replica sets* that provide your shards.
   - the `mongod` instance that provides your existing config database.
   - all `mongos` (page 676) instances in your cluster.
2. Copy the entire `dbpath` (page 624) 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 Contents* (page 675), `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` (page 676) processes.

Migrate Config Servers with the Same Hostname

Use this process when you need to migrate a config server to a new system but the new system will be accessible using the same host name.

1. Shut down the config server that you're moving.

   This will render all config data for your cluster *read only* (page 110).

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. Move the entire `dbpath` (page 624) file system tree from the old config server to the new config server. This command, issued on the old config server system, may resemble the following:

   ```
   rsync -az /data/configdb mongo-config0.example.net:/data/configdb
   ```

4. Start the config 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

Use this process when you need to migrate a *Config Database Contents* (page 675) to a new server and it *will not* be accessible via the same hostname. If possible, avoid changing the hostname so that you can use the *previous procedure* (page 127).

1. Shut down the *config server* (page 110) you're moving.

   This will render all config data for your cluster "read only:"

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```

2. Start the config instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

3. Shut down all existing MongoDB processes. This includes:

   - all `mongod` instances or *replica sets* that provide your shards.
   - the `mongod` instances that provide your existing *config databases* (page 675).
   - all `mongos` (page 676) instances in your cluster.

4. Restart all `mongod` processes that provide the shard servers.

5. Update the *`--configdb`* (page 590) parameter (or `configdb` (page 630)) for all `mongos` (page 676) instances and restart all `mongos` (page 676) instances.

Replace a Config Server

Use this procedure only if you need to replace one of your config servers after it becomes 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 process for *migrating a config server to a different hostname* (page 128).

1. 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.

2. Shut down *one* (and only one) of the existing config servers. Copy all this host's `dbpath` (page 624) 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
   ```

3. Restart the config server process that you used in the previous step to copy the data files to the new config server instance.

4. Start the new config server instance. The default invocation is:

   ```
   mongod --configsvr
   ```

Note: In the course of this procedure *never* remove a config server from the `configdb` (page 630) parameter on any of the `mongos` (page 676) instances. If you need to change the name of a config server, always make sure that all `mongos` (page 676) instances have three config servers specified in the `configdb` (page 630) setting at all times.

Backup Cluster Metadata

The cluster will remain operational [4] without one of the config database's `mongod` instances, creating a backup of the cluster metadata from the config database is straight forward:

1. Shut down one of the *config databases*.
2. Create a full copy of the data files (i.e. the path specified by the `dbpath` (page 624) option for the config instance.)
3. Restart the original configuration server.

See Also:

Backup and Restoration Strategies (page 180).

8.2.6 Troubleshooting

The two most important factors in maintaining a successful sharded cluster are:

- *choosing an appropriate shard key* (page 133) and
- *sufficient capacity to support current and future operations* (page 108).

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.

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 137). While the default chunk size is configurable with the `chunkSize` (page 630) 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 137) 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 120).

Finally, if your shard key has a low *cardinality* (page 133), MongoDB may not be able to create sufficient splits among the data.

One Shard Receives too much Traffic

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 134).

[4] While one of the three config servers unavailable, no the cluster cannot split any chunks nor can it migrate chunks between shards. Your application will be able to write data to the cluster. The *Config Servers* (page 110) section of the documentation provides more information on this topic.

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 135) to correct this pattern.

The Cluster does not Balance

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 129).

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 133) 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 125) that is too short, given the rate of data growth.
  - an uneven distribution of *write operations* (page 134) 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.

Migrations Render Cluster Unusable

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 125) 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 122) 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 117) 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 134).

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 180), 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 125) 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 126) for the duration of the backup procedure.

Confirm that the balancer is not active using the `sh.getBalancerState()` method before starting a backup operation. When the backup procedure is complete you can reactivate the balancer process.

8.3 Sharded Cluster Architectures

This document describes the organization and design of *sharded cluster* deployments. For documentation of common administrative tasks related to sharded clusters, see *Sharded Cluster Administration* (page 113). For complete documentation of sharded clusters see the *Sharding* (page 105) section of this manual.

See Also:

Sharding Requirements (page 108).

8.3.1 Deploying a Test Cluster

Warning: Use this architecture for testing and development only.

You can deploy a very minimal cluster for testing and development. These *non-production* clusters have the following components:

- 1 *config server* (page 110).
- At least one `mongod` instance (either *replica sets* or as a standalone node.)
- 1 `mongos` (page 676) instance.

8.3.2 Deploying a Production Cluster

When deploying a production cluster, you must ensure that the data is redundant and that your systems are highly available. To that end, a production-level cluster must have the following components:

- 3 *config servers* (page 110), each residing on a discrete system.

  Note: A single *sharded cluster* must have exclusive use of its *config servers* (page 110). If you have multiple shards, you will need to have a group of config servers for each cluster.

- 2 or more *replica sets*, for the *shards*.

  See Also:

  For more information on replica sets see *Replication Architectures* (page 51) and *Replication* (page 31).

- `mongos` (page 676) instances. Typically, you will deploy a single `mongos` (page 676) instance on each application server. Alternatively, you may deploy several *mongos* nodes and let your application connect to these via a load balancer.

See Also:

Add a Shard to a Cluster (page 117) and *Remove a Shard from a Cluster* (page 117).

8.3.3 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. [5] However, in production deployments some databases and collections will use sharding, while other databases and collections will only reside on a single database instance or replica set (i.e. a *shard*.)

Note: 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` (page 676) even for operations that do not impact the sharded data.

Every database has a "primary" [6] shard that holds all un-sharded collections in that database. All collections that *are not* sharded reside on the primary for their database. Use the `movePrimary` command to change the primary shard for a database. Use the `printShardingStatus` command or the `sh.status()` to see an overview of the cluster, which contains information about the *chunk* and database distribution within the cluster.

Warning: The `movePrimary` command can be expensive because it copies all non-sharded data to the new shard, during which that data will be unavailable for other operations.

When you deploy a new *sharded cluster*, the "first shard" becomes the primary for all databases before enabling sharding. Databases created subsequently, may reside on any shard in the cluster.

8.3.4 High Availability and MongoDB

A *production* (page 131) *cluster* has no single point of failure. This section introduces the availability concerns for MongoDB deployments, and highlights potential failure scenarios and available resolutions:

- Application servers or `mongos` (page 676) instances become unavailable.

  If each application server has its own `mongos` (page 676) instance, other application servers can continue access the database. Furthermore, `mongos` (page 676) instances do not maintain persistent state, and they can restart and become unavailable without loosing any state or data. When a `mongos` (page 676) 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 31) provide high availability for shards. If the unavailable `mongod` is a *primary*, then the replica set will *elect* (page 34) a new primary. If the unavailable `mongod` is a *secondary*, and it connects within its *recovery window* (page 36). 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.

  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 on 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 database* become unavailable.

[5] 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` on a collection within that database.

[6] The term "primary" in the context of databases and sharding, has nothing to do with the term *primary* in the context of *replica sets*.

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 112) and the cluster can create no new *chunk splits* (page 120). 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*.

8.4 Sharding Internals

This document introduces lower level sharding concepts for users who are familiar with *sharding* generally and want to learn more about the internals. This document provides a more detailed understanding of your cluster's behavior. For higher level sharding concepts, see *Sharding Fundamentals* (page 107). For complete documentation of sharded clusters see the *Sharding* (page 105) section of this manual.

8.4.1 Shard Keys

Shard keys are the field in a collection that MongoDB uses to distribute *documents* within a sharded cluster. See the *overview of shard keys* (page 109) for an introduction to these topics.

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-splitable 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-splitable.

  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 134) or appropriate *write scaling* (page 134). Please continue reading for more information on these topics.

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*.

`ObjectID` is computed upon document creation, that is 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 133), 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 134), which is another important characteristic of shard keys.

Querying

The `mongos` (page 676) provides an interface for applications to interact with sharded clusters that hides the complexity of *data partitioning*. A `mongos` (page 676) receives queries from applications, and uses metadata from the *config server* (page 110), to route queries to the `mongod` instances with the appropriate data. While the `mongos` (page 676) 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 *mongos and Sharding* (page 111) and *config server* (page 110) sections for a more general overview of querying in sharded environments.

Query Isolation

The fastest queries in a sharded environment are those that `mongos` (page 676) will route to a single shard, using the *shard key* and the cluster meta data from the *config server* (page 110). For queries that don't include the shard key, `mongos` (page 676) 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 [7], the `mongos` (page 676) 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` (page 676) 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.

[7] 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 Indexes wiki page for more information on indexes and compound indexes.

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 splitable with a compound shard key.

See Also:

mongos and Querying (page 111) for more information on query operations in the context of sharded clusters. Specifically the *Routing* (page 111) sub-section outlines the procedure that `mongos` (page 676) uses to route read operations to the shards.

Sorting

If you use the `sort()` method on a query in a sharded MongoDB environment *and* the sort is on a field that is *not* part of the shard key, `mongos` (page 676) must send the query to all `mongod` instances in the cluster. `mongos` (page 676) must wait for a response from every shard before it can merge the results and return data. If you require high performance sorted queries, ensure that the sort key is a component of the shard key.

Operations and Reliability

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` (page 676) 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` (page 676) 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.

Choosing a Shard Key

It is unlikely that any single, naturally occurring key in your collection will satisfy all requirements of a good shard key. There are three options:

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
   - query patterns and demands.

From a decision making stand point, begin by finding the field that will provide the required *query isolation* (page 134), ensure that *writes will scale across the cluster* (page 134), and then add an additional field to provide additional *cardinality* (page 133) if your primary key does not have sufficient split-ability.

Shard Key Indexes

All sharded collections **must** have an index that starts with the *shard key*. If you shard a collection that does not yet contain documents and *without* such an index, the `shardCollection` will create an index on the shard key. If the collection already contains documents, you must create an appropriate index before using `shardCollection`. Changed in version 2.2: The index on the shard key no longer needs to be identical to the shard key. This index can be an index of the shard key itself as before, or a *compound index* where the shard key is the prefix of the index. This index *cannot* be a multikey index. If you have a collection named `people`, sharded using the field `{ zipcode: 1 }`, and you want to replace this with an index on the field `{ zipcode: 1, username: 1 }`, then:

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 existing index on `{ zipcode: 1 }`:

   ```
   db.people.dropIndex( { zipcode: 1 } );
   ```

Warning: The index on the shard key **cannot** be a multikey index.
As above, an index on `{ zipcode: 1, username: 1 }` can only replace an index on `zipcode` if there are no array values for the `username` field.
If you drop the last appropriate index for the shard key, recover by recreating a index on just the shard key.

8.4.2 Cluster Balancer

The *balancer* (page 112) sub-process is responsible for redistributing chunks evenly among the shards and ensuring that each member of the cluster is responsible for the same volume of data. This section contains complete documentation of the balancer process and operations. For a higher level introduction see the *Balancing and Distribution* (page 112) section.

Balancing Internals

A balancing round originates from an arbitrary `mongos` (page 676) instance from one of the cluster's `mongos` (page 676) instances. When a balancer process is active, the responsible `mongos` (page 676) acquires a "lock" by modifying a document in the `lock` collection in the *Config Database Contents* (page 675).

By default, the balancer process is always running. When the number of chunks in a collection is unevenly distributed among the shards, the balancer begins migrating *chunks* from shards with more chunks to shards with a fewer number of chunks. The balancer will continue migrating chunks, one at a time, until the data is evenly distributed among the shards.

While these automatic chunk migrations are crucial for distributing data, they carry some overhead in terms of bandwidth and workload, both of which can impact database performance. As a result, MongoDB attempts to minimize the effect of balancing by only migrating chunks when the distribution of chunks passes the *migration thresholds* (page 137).

The migration process ensures consistency and maximizes availability of chunks during balancing: when MongoDB begins migrating a chunk, the database begins copying the data to the new server and tracks incoming write operations. After migrating chunks, the "from" `mongod` sends all new writes to the "receiving" server. Finally, `mongos` (page 676) updates the chunk record in the *config database* to reflect the new location of the chunk.

Note: Changed in version 2.0: Before MongoDB version 2.0, large differences in timekeeping (i.e. clock skew) between `mongos` (page 676) instances could lead to failed distributed locks, which 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.

Migration Thresholds

Changed in version 2.2: The following thresholds appear first in 2.2; prior to this release, balancing would only commence if the shard with the most chunks had 8 more chunks than the shard with the least number of chunks. In order to minimize the impact of balancing on the cluster, the *balancer* will not begin balancing until the distribution of chunks has reached certain thresholds. These thresholds apply to the difference in number of *chunks* between the shard with the greatest number of chunks and the shard with the least number of chunks. The balancer has the following thresholds:

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 the difference between the number of chunks on any two shards is *less than two.*

Note: You can restrict the balancer so that it only operates between specific start and end times. See *Schedule the Balancing Window* (page 125) for more information.

The specification of the balancing window is relative to the local time zone of all individual `mongos` (page 676) instances in the sharded cluster.

Chunk Size

The default *chunk* size in MongoDB is 64 megabytes.

When chunks grow beyond the *specified chunk size* (page 137) a `mongos` (page 676) instance will split the chunk in half. This will eventually lead to migrations, when chunks become unevenly distributed among the cluster. The `mongos` (page 676) instances will initiate a round of migrations to redistribute data in the cluster.

Chunk size is arbitrary and must account for the following:

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations, which creates expense at the query routing (`mongos` (page 676)) layer.
2. Large chunks lead to fewer migrations, which is more efficient both from the networking perspective *and* in terms internal overhead at the query routing layer. Large chunks produce these efficiencies 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, but this value is *configurable* (page 122). Be aware of the following limitations when modifying chunk size:

- Automatic splitting only occurs when inserting *documents* or updating existing documents; 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 insertion or updates until they reach the new size.

Shard Size

By default, MongoDB will attempt to fill all available disk space with data on every shard as the data set grows. Monitor disk utilization in addition to other performance metrics, to ensure that the cluster always has capacity to accommodate additional data.

You can also configure a "maximum size" for any shard when you add the shard using the `maxSize` parameter of the `addShard` command. This will prevent the *balancer* from migrating chunks to the shard when the value of `mem.mapped` (page 642) exceeds the `maxSize` setting.

See Also:

Monitoring Database Systems (page 170).

Chunk Migration

MongoDB migrates chunks in a *sharded cluster* to distribute data evenly among shards. Migrations may be either:

- Manual. In these migrations you must specify the chunk that you want to migrate and the destination shard. Only migrate chunks manually after initiating sharding, to distribute data during bulk inserts, or if the cluster becomes uneven. See *Migrating Chunks* (page 123) for more details.
- Automatic. The balancer process handles most migrations when distribution of chunks between shards becomes uneven. See *Migration Thresholds* (page 137) for more details.

All chunk migrations use the following procedure:

1. The balancer process sends the `moveChunk` command to the source shard for the chunk. In this operation the balancer passes the name of the destination shard to the source shard.
2. The source initiates the move with an internal `moveChunk` command with the destination shard.
3. The destination shard begins requesting documents in the chunk, and begins receiving these chunks.
4. After receiving the final document in the chunk, the destination shard initiates a synchronization process to ensure that all changes to the documents in the chunk on the source shard during the migration process exist on the destination shard.

   When fully synchronized, the destination shard connects to the *config database* and updates the chunk location in the cluster metadata. After completing this operation, once there are no open cursors on the chunk, the source shard starts deleting its copy of documents from the migrated chunk.

When the `_secondaryThrottle` is `true` for `moveChunk` or the *balancer*, MongoDB ensure that *one secondary* member has replicated changes before allowing new chunk migrations.

Detect Connections to `mongos` Instances

If your application must detect if the MongoDB instance its connected to is `mongos` (page 676), use the `isMaster` command. When a client connects to a `mongos` (page 676), `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.

8.4.3 Config Database

The `config` database contains information about your sharding configuration and stores the information in a set of collections used by sharding.

Important: Back up the `config` database before performing 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* manipulate the content of the config database directly. The `config` database contains the following collections:

- `chunks` (page 675)
- `collections` (page 652)
- `databases` (page 676)
- `lockpings` (page 676)
- `locks` (page 671)
- `mongos` (page 676)
- `settings` (page 664)
- `shards` (page 677)
- `version` (page 638)

See *Config Database Contents* (page 675) for full documentation of these collections and their role in sharded clusters.

CHAPTER

NINE

TUTORIALS

The following tutorials describe specific sharding procedures:

9.1 Deploy a Sharded Cluster

This document describes how to deploy a *sharded cluster* for a standalone `mongod` instance. To deploy a cluster for an existing replica set, see *Convert a Replica Set to a Replicated Sharded Cluster* (page 195).

9.1.1 Procedure

Before deploying a sharded cluster, see the requirements listed in *Requirements for Sharded Clusters* (page 108).

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 *`mongos --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 database processes are small `mongod` instances that store the cluster's metadata. You must have exactly *three* instances in production deployments. Each stores a complete copy of the cluster's metadata. These instances should run on different servers to assure good uptime and data safety.

Since config database `mongod` instances receive relatively little traffic and demand only a small portion of system resources, you can run the instances on systems that run other cluster components.

By default a `mongod` *`--configsvr`* (page 587) process stores its data files in the */data/configdb* directory. You can specify a different location using the `dbpath` (page 624) run-time option. The config `mongod` instance is accessible via port `27019`. In addition to `configsvr` (page 629), use other `mongod` *runtime options* (page 621) as needed.

To create a data directory for each config server, issue a command similar to the following for each:

```
mkdir /data/db/config
```

To start each config server, issue a command similar to the following for each:

```
mongod --configsvr --dbpath <path> --port <port>
```

Start the mongos Instances

The mongos (page 676) instance routes queries and operations to the appropriate shards and interacts with the config server instances. All client operations targeting a cluster go through mongos (page 676) instances.

mongos (page 676) instances are lightweight and do not require data directories. A cluster typically has several instances. For example, you might run one mongos (page 676) instance on each of your application servers, or you might run a mongos (page 676) instance on each of the servers running a mongod process.

You must the specify resolvable hostnames [1] for the *3* config servers when starting the mongos (page 676) instance. You specify the hostnames either in the configuration file or as command line parameters.

The mongos (page 676) instance runs on the default MongoDB TCP port: 27017.

To start mongos (page 676) instance running on the mongos0.example.net host, that connects to the config server instances running on the following hosts:

- mongoc0.example.net
- mongoc1.example.net
- mongoc2.example.net

You would issue the following command:

```
mongos --configdb mongoc0.example.net,mongoc1.example.net,mongoc2.example.net
```

Add Shards to the Cluster

You must deploy at least one *shard* or one *replica set* to begin. In a production cluster, each shard is a replica set. You may add additional shards to a running cluster later. For instructions on deploying replica sets, see *Deploy a Replica Set* (page 73).

This procedure assumes you have two active and initiated replica sets and describes how to add the first two shards to the cluster.

First, connect to one of the mongos (page 676) instances. For example, if a mongos (page 676) is accessible at mongos0.example.net on port 27017, issue the following command:

```
mongo mongos0.example.net
```

Then, from the mongo shell connected to the mongos (page 676) instance, call the sh.addShard() method for each shard that you want to add to the cluster:

```
sh.addShard( "sfo30.example.net" )
sh.addShard( "sfo40.example.net" )
```

If sfo30.example.net and sfo40.example.net are members of a replica set, MongoDB will discover all other members of the replica set.

These operations add two shards, provided by:

- the replica set named s0, that includes the sfo30.example.net host.
- the replica set name and s1, that includes the sfo40.example.net host.

All shards should be replica sets Changed in version 2.0.3. After version 2.0.3, you may use the above form to add replica sets to a cluster. The cluster will automatically discover the other members of the replica set and note their names accordingly.

[1] Use DNS names for the config servers rather than explicit IP addresses for operational flexibility. If you're not using resolvable hostname, you cannot change the config server names or IP addresses without a restarting *every* mongos (page 676) and mongod instance.

Before version 2.0.3, you must specify the shard in the following form: the replica set name, followed by a forward slash, followed by a comma-separated list of seeds for the replica set. For example, if the name of the replica set is `sh0`, and the replica set were to have three members, then your `sh.addShard` command might resemble:

```
sh.addShard( "sh0/sfo30.example.net,sfo31.example.net,sfo32.example.net" )
```

The `sh.addShard()` helper in the `mongo` shell is a wrapper for the `addShard` *database command*.

Enable Sharding for Databases

While sharding operates on a per-collection basis, you must enable sharding for each database that holds collections you want to shard. A single cluster may have many databases, with each database housing collections.

Use the following operation in a `mongo` shell session connected to a `mongos` (page 676) instance in your cluster:

```
sh.enableSharding("records")
```

Where `records` is the name of the database that holds the collection you want to shard. `sh.enableSharding()` is a wrapper around the `enableSharding` *database command*. You can enable sharding for multiple databases in the cluster.

Enable Sharding for Collections

You can enable sharding on a per-collection basis. Because MongoDB uses "range based sharding," you must specify the *shard key* MongoDB uses to distribute your documents among the shards. For more information, see the *overview of shard keys* (page 109).

To enable sharding for a collection, use the `sh.shardCollection()` helper in the `mongo` shell. The helper provides a wrapper around the `shardCollection` *database command* and has the following prototype form:

```
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.

Consider the following example invocations of `sh.shardCollection()`:

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
sh.shardCollection("events.alerts", { "hashed_id": 1 } )
```

In order, these operations shard:

1. 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 *splitable* (page 133) by the values of the `name` field.

2. 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 *splitable* (page 133) by the values of the `_id` field.

3. 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 *splitable* (page 133) by the values of the `_id` field.

4. The `alerts` collection in the `events` database using the shard key `{ "hashed_id": 1 }`.

   This shard key distributes documents by the value of the `hashed_id` field. Presumably this is a computed value that holds the hash of some value in your documents and is able to evenly distribute documents throughout your cluster.

9.2 Add Shards to an Existing Cluster

9.2.1 Synopsis

This document describes how to add a *shard* to an existing *sharded cluster*. As your data set grows you must add additional shards to a cluster to provide additional capacity. For additional sharding procedures, see *Sharded Cluster Administration* (page 113).

9.2.2 Concerns

Distributing *chunks* among your cluster requires some capacity to support the migration process. When adding a shard to your cluster, you should always ensure that your cluster has enough capacity to support the migration without affecting legitimate production traffic.

In production environments, all shards should be *replica sets*. Furthermore, *all* interaction with your sharded cluster should pass through a `mongos` (page 676) instance. This tutorial assumes that you already have a `mongo` shell connection to a `mongos` (page 676) instance.

9.2.3 Process

Tell the cluster where to find the individual shards. You can do this using the `addShard` command:

```
db.runCommand( { addShard: mongodb0.example.net, name: "mongodb0" } )
```

Or you can use the `sh.addShard()` helper in the `mongo` shell:

```
sh.addShard( "[hostname]:[port]" )
```

Replace `[hostname]` and `[port]` with the hostname and TCP port number of where the shard is accessible.

For example:

```
sh.addShard( "mongodb0.example.net:27027" )
```

If `mongodb0.example.net:27027` is a member of a replica set, call the `sh.addShard()` method with an argument that resembles the following:

```
sh.addShard( "<setName>/mongodb0.example.net:27027" )
```

Replace, `<setName>` with the name of the replica set, and MongoDB will discover all other members of the replica set.

Note: In production deployments, all shards should be replica sets. Changed in version 2.0.3. Before version 2.0.3, you must specify the shard in the following form:

```
replicaSetName/<seed1>,<seed2>,<seed3>
```

For example, if the name of the replica set is `repl0`, then your `sh.addShard` command would be:

```
sh.addShard( "repl0/mongodb0.example.net:27027,mongodb1.example.net:27017,mongodb2.example.net:27017"
```

Repeat this step for each shard in your cluster.

Optional

You may specify a "name" as an argument to the `addShard`, follows:

```
db.runCommand( { addShard: mongodb0.example.net, name: "mongodb0" } )
```

You cannot specify a name for a shard using the `sh.addShard()` helper in the `mongo` shell. If you use the helper or do not specify a shard name, then MongoDB will assign a name upon creation.

Note: It may take some time for *chunks* to migrate to the new shard because the system must copy data from one `mongod` instance to another while maintaining data consistency.

For an overview of the balancing operation, see the *Balancing and Distribution* (page 112) section.

For additional information on balancing, see the *Balancing Internals* (page 136) section.

9.3 Remove Shards from an Existing Sharded Cluster

9.3.1 Synopsis

This procedure describes the procedure for migrating data from a shard safely, when you need to decommission a shard. You may also need to remove shards as part of hardware reorganization and data migration.

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, you will:

- Move *chunks* off of the shard.
- Ensure that this shard is not the *primary shard* for any databases in the cluster. If it is, move the "primary" status for these databases to other shards.
- Remove the shard from the cluster's configuration.

9.3.2 Procedure

Complete this procedure by connecting to any `mongos` (page 676) in the cluster using the `mongo` shell.

You can only remove a shard by its shard name. To discover or confirm the name of a shard, use the `listShards` command, `printShardingStatus` command, or `sh.status()` shell helper.

The example commands in this document remove a shard named `mongodb0`.

Note: To successfully migrate data from a shard, the *balancer* process **must** be active. Check the balancer state using the `sh.getBalancerState()` helper in the `mongo` shell. For more information, see the section on *balancer operations* (page 126).

Remove Chunks from the Shard

Start by running the `removeShard` command. This begins "draining" chunks from the shard you are removing.

```
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 in your cluster, 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, run `removeShard` again at any stage of the process, as follows:

```
db.runCommand( { removeshard: "mongodb0" } )
```

The output resembles the following document:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 42, dbs : 1 }, ok: 1 }
```

In the `remaining` sub document, a counter 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 Databases

Databases with non-sharded collections store those collections on a single shard known as the *primary shard* for that database. The following step is necessary only when the shard to remove is also the primary shard for one or more databases.

Issue the following command at the `mongo` shell:

```
db.runCommand( { movePrimary: "myapp", to: "mongodb1" })
```

This command migrates all remaining non-sharded data in the database named `myapp` to the shard named `mongodb1`.

Warning: Do not run the `movePrimary` until you have *finished* draining the shard.

This command will 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

Run `removeShard` again to clean up all metadata information and finalize the removal, as follows:

```
db.runCommand( { removeshard: "mongodb0" } )
```

A success message appears at completion:

```
{ msg: "remove shard completed successfully" , stage: "completed", host: "mongodb0", ok : 1 }
```

When the value of "state" is "completed", you may safely stop the `mongodb0` shard.

9.4 Enforce Unique Keys for Sharded Collections

9.4.1 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* (page 680) because insert and indexing operations are local to each shard. [2]

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 109).

   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.

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.

Regardless of method, be aware that writes to the MongoDB database are "fire and forget," or "unsafe" by default: they will not return errors to the client if MongoDB rejects a write operation because of a duplicate key or other error. As a result if you want to enforce unique keys you **must** use the safe write setting in your driver. See your *driver's documentation* (page 285) on `getLastError` for more information.

9.4.2 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:

[2] 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*.

```
db.runCommand( { shardCollection : "test.users" } )
```

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 134) and *query isolation* (page 134), as well as *high cardinality* (page 133). These ideal shard keys are not often the same keys that require uniqueness and requires a different approach.

9.4.3 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 147); 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* (page 591):

```
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 Also:

The full documentation of: `db.collection.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.

9.5 Convert a Replica Set to a Replicated Sharded Cluster

9.5.1 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.

9.5.2 Process

Install MongoDB according to the instructions in the *MongoDB Installation Tutorial* (page 9).

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 197).

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:

   - `http://docs.mongodb.org/manual/data/example/firstset1`
   - `http://docs.mongodb.org/manual/data/example/firstset2`
   - `http://docs.mongodb.org/manual/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* (page 586) option restricts the size of the operation log (i.e. oplog) for each `mongod` instance to 700MB. Without the *--oplogSize* (page 586) 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":
                               }
```

   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:

   - `http://docs.mongodb.org/manual/data/example/config1`
   - `http://docs.mongodb.org/manual/data/example/config2`
   - `http://docs.mongodb.org/manual/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` (page 676) 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` (page 590) (1MB works well.) The default `chunkSize` (page 630) 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 (page 630) options specify the *configuration databases* (e.g. localhost:20001, localhost:20002, and localhost:2003). The mongos (page 676) 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* (page 589) option, as 27017 is the default port.

4. Add the first shard in mongos (page 676). In a new terminal window or GNU Screen session, add the first shard, according to the following procedure:

   (a) Connect to the mongos (page 676) 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:
   - http://docs.mongodb.org/manual/data/example/secondset1
   - http://docs.mongodb.org/manual/data/example/secondset2
   - http://docs.mongodb.org/manual/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 (page 628) 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` (page 676) 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 109) and *Shard Key* (page 133) sections.

Shard the Collection

Issue the following command:

```
use admin
db.runCommand( { shardCollection : "test.test_collection", key : {"number":1} })
{ "collectionsharded" : "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` (page 676) 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.

CHAPTER

TEN

REFERENCE

The following reference section describes sharding commands:

- *sharding-commands*

Part V

Administration

This page lists the core administrative documentation and the administration tutorials. This page also provides links to administrative documentation for replica sets, sharding, and indexes.

CHAPTER

ELEVEN

CORE COMPETENCIES

The following documents outline basic MongoDB administrative topics:

11.1 Run-time Database Configuration

The *command line* (page 581) and *configuration file* (page 621) 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 the same collection of options and settings, this document primarily uses the configuration file interface. If you run MongoDB using a control script or packaged for your operating system, you likely already have a configuration file located at `http://docs.mongodb.org/manual/etc/mongodb.conf`. Confirm this by checking the content of the `http://docs.mongodb.org/manual/etc/init.d/mongod` or `http://docs.mongodb.org/manual/etc/rc.d/mongod` script to insure that the *control scripts* start the `mongod` with the appropriate configuration file (see below.)

To start 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 `http://docs.mongodb.org/manual/etc/mongodb.conf` file on your system to control the configuration of your database instance.

11.1.1 Starting, Stopping, and Running 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` (page 623)" 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 (page 622) 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*) systems.
- port (page 622) 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 1000.

- quiet (page 622) 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 (page 624) is http://docs.mongodb.org/manual/srv/mongodb, which specifies where MongoDB will store its data files. http://docs.mongodb.org/manual/srv/mongodb and http://docs.mongodb.org/manual/var/lib/mongodb are popular locations. The user account that mongod runs under will need read and write access to this directory.
- logpath (page 622) is http://docs.mongodb.org/manual/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 (page 623) is true, which ensures that mongod does not overwrite an existing log file following the server start operation.
- **journal (page 625) 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.

11.1.2 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 (page 622)" 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 (page 623)" is true which disables the UNIX Socket, which is otherwise enabled by default. This limits access on the local system. This is desirable when running MongoDB on with shared access, but in most situations has minimal impact.
- "auth (page 624)" is true which 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:

The "Security and Authentication" wiki page.

11.1.3 Replication and Sharding Configuration

Replication Configuration

Replica set configuration is straightforward, and only requires that the `replSet` (page 628) have a value that is consistent among all members of the set. Consider the following:

```
replSet = set0
```

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

See Also:

"*Replica set reconfiguration* (page 665).

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` (page 623) 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` (page 676) instances that connect to the set. The keyfile must be less one kilobyte in size and may only contain characters in the base64 set and file must not have group or "world" permissions on UNIX systems.

See Also:

The "*Replica set Reconfiguration* (page 665) section for information regarding the process for changing replica set during operation.

Additionally, consider the "*Replica Set Security* (page 46)" section for information on configuring authentication with replica sets.

Finally, see the "*Replication* (page 31)" index and the "*Replication Fundamentals* (page 33)" 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 163) `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` (page 676)" and "`mongod`" instances.

To set up shards, configure two or more mongod instance using your *base configuration* (page 163), adding the shardsvr (page 629) setting:

```
shardsvr = true
```

Finally, to establish the cluster, configure at least one mongos (page 676) process with the following settings:

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

You can specify multiple configdb (page 630) instances by specifying hostnames and ports in the form of a comma separated list. In general, avoid modifying the chunkSize (page 630) from the default value of 64, [1] and *should* ensure this setting is consistent among all mongos (page 676) instances.

See Also:

The "*Sharding* (page 105)" section of the manual for more information on sharding and cluster configuration.

11.1.4 Running Multiple Database Instances on the Same System

In many cases running multiple instances of mongod on a single system is not recommended, on some types of deployments [2] 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 163) for each instance, but consider the following configuration values:

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

The dbpath (page 624) 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 (page 623) controls where mongod process places it's *pid* 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.

11.1.5 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 163) and add these options if you are experiencing some unknown issue or performance problem as needed:

- slowms (page 627) 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 the "Optimization" wiki page for more information on optimizing operations in MongoDB.

[1] *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.

[2] 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.

- `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` (page 621) 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` (page 624) enables *diagnostic logging*. Level `3` logs all read and write options.
- `objcheck` (page 622) 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` (page 624) 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.

11.2 Using MongoDB with SSL Connections

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 MongoDB does **not** contain support for SSL.

As of the current release, to use SSL you must either: build MongoDB locally passing the "`--ssl`" option to `scons`, or use the MongoDB subscriber build.

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.

11.2.1 `mongod` and `mongos` SSL Configuration

Add the following command line options to your `mongod` invocation:

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem> --sslPEMKeyPassword <pass>
```

Replace "`<pem>`" with the path to your SSL certificate `.pem` file, and "`<pass>`" with the password you used to encrypt the `.pem` file.

You may also specify these options in your "`mongodb.conf`" file with following options:

```
sslOnNormalPorts = true
sslPEMKeyFile = /etc/ssl/mongodb.pem
sslPEMKeyPassword = pass
```

Modify these values to reflect the location of your actual `.pem` file and its password.

You can specify these configuration options in a configuration file for `mongos` (page 676), or start `mongos` (page 676) with the following invocation:

```
mongos --sslOnNormalPorts --sslPEMKeyFile <pem> --sslPEMKeyPassword <pass>
```

You can use any existing SSL certificate, or you can generate your own SSL certificate using a command that resembles the following:

```
cd /etc/ssl/
openssl req -new -x509 -days 365 -nodes -out mongodb-cert.pem -keyout mongodb-cert.key
```

To create the combined `.pem` file that contains the `.key` file and the `.pem` certificate, use the following command:

```
cat mongodb-cert.key mongodb-cert.pem > mongodb.pem
```

11.2.2 Clients

Clients must have support for SSL to work with a `mongod` instance that has SSL support enabled. The current versions of the Python, Java, Ruby, and Node.js drivers have support for SSL, with full support coming in future releases of other drivers.

mongo

The `mongo` shell built with ssl support distributed with the subscriber build also supports SSL. Use the "`--ssl`" flag as follows:

```
mongo --ssl --host <host>
```

MMS

The MMS 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 and is currently enabled on a group by group basis by 10gen.

Please see the MMS Manual for more information about MMS configuration.

PyMongo

Add the "`ssl=True`" parameter to a PyMongo py:module:*connection <pymongo:pymongo.connection>* to create a MongoDB connection to an SSL MongoDB instance:

```
from pymongo import Connection
c = Connection(host="mongodb.example.net", port=27017, ssl=True)
```

To connect to a replica set, use the following operation:

```
from pymongo import ReplicaSetConnection
c = ReplicaSetConnection("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.SSLContext;

public class sslApp {

public static void main(String args[])
     throws Exception {

     MongoOptions o = new MongoOptions();
     o.socketFactory = SSLSocketFactory.getDefault();

    Mongo m = new Mongo( "localhost" , o );

    DB db = m.getDB( "test" );
     DBCollection c = db.getCollection( "foo" );

    System.out.println( c.findOne() );

}
}
```

Ruby

The recent versions version 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 = Mongo::Connection.new('localhost', 27017, :ssl => true)
```

Replace `connection` with the following if you're connecting to a replica set:

```
connection = Mongo::ReplSetConnection.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 driver, use the following invocation to connect to a `mongod` or `mongos` (page 676) 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` (page 676) 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);
```

11.3 Monitoring Database Systems

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 issues as you encounter them, rather than waiting for a crisis or failure.

This document provides an overview of the available tools and data provided by MongoDB as well as introduction to diagnostic strategies, and suggestions for monitoring instances in MongoDB's replica sets and sharded clusters.

Note: 10gen provides a hosted monitoring service which collects and aggregates these data to provide insight into the performance and operation of MongoDB deployments. See the MongoDB Monitoring Service (MMS) and the MMS documentation for more information.

11.3.1 Monitoring Tools

There are two primary methods for collecting data regarding the state of a running MongoDB instance. First, there are a set of tools distributed with MongoDB that provide real-time reporting of activity on the database. Second, several `database commands` return statistics regarding the current database state with greater fidelity. Both methods allow you to collect data that answers a different set of questions, and are useful in different contexts.

This section provides an overview of these utilities and statistics, along with an example of the kinds of questions that each method is most suited to help you address.

Utilities

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

mongotop

`mongotop` tracks and reports the current read and write activity of a MongoDB instance. `mongotop` provides per-collection visibility into use. Use `mongotop` to verify that activity and use match expectations. See the *mongotop manual* (page 615) for details.

mongostat

`mongostat` captures and returns counters of database operations. `mongostat` reports operations on a per-type (e.g. insert, query, update, delete, etc.) basis. This format makes it easy to understand the distribution of load on the server. Use `mongostat` to understand the distribution of operation types and to inform capacity planning. See the *mongostat manual* (page 611) for details.

REST Interface

MongoDB provides a *REST* interface that exposes a diagnostic and monitoring information in a simple web page. Enable this by setting `rest` (page 626) to `true`, and access this page via the local host interface using the port numbered 1000 more than that the database port. In default configurations the REST interface is accessible on `28017`. For example, to access the REST interface on a locally running mongod instance: http://localhost:28017

Statistics

The `mongo` shell provides a number of commands that return statistics about the state of the MongoDB instance. These data may provide finer granularity regarding the state of the MongoDB instance than the tools above. Consider using their output in scripts and programs to develop custom alerts, or modifying the behavior of your application in response to the activity of your instance.

serverStatus

Access *serverStatus data* (page 637) by way of the `serverStatus` command. This *document* contains a general overview of the state of the database, including disk usage, memory use, connection, journaling, index accesses. The command returns quickly and does not impact MongoDB performance.

While this output contains a (nearly) complete account of the state of a MongoDB instance, in most cases you will not run this command directly. Nevertheless, all administrators should be familiar with the data provided by `serverStatus`.

See Also:

`db.stats()` and *serverStatus data* (page 637).

replSetGetStatus

View the *replSetGetStatus data* (page 659) with the `replSetGetStatus` command (`rs.status()` from the shell). The document returned by this command reflects the state and configuration of the replica set. Use this data to ensure that replication is properly configured, and to check the connections between the current host and the members of the replica set.

dbStats

The *dbStats data* (page 651) is accessible by way of the `dbStats` command (`db.stats()` from the shell). This command returns a document that contains data that reflects the amount of storage used and data contained in the database, as well as object, collection, and index counters. Use this data to check and track the state and storage of a specific database. This output also allows you to compare utilization between databases and to determine average *document* size in a database.

collStats

The *collStats data* (page 653) is accessible using the `collStats` command (`db.printCollectionStats()` from the shell). It provides statistics that resemble `dbStats` on the collection level: this includes 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 the indexes.

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, usually open source.

Tool	Plugin	Description
Ganglia	mongodb-ganglia	Shell script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	gmond_python_modul	Parses output from the `serverStatus` and `replSetGetStatus` commands.
Motop	*None*	Realtime monitoring tool for several MongoDB servers. Shows current operations ordered by durations every second.
mtop	*None*	A top like tool.
Munin	mongo-munin	Retrieves server statistics.
Munin	mongomon	Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).
Munin	munin-plugins Ubuntu PPA	Some additional munin plugins not in the main distribution.
Nagios	nagios-plugin-mongodb	A simple Nagios check script.
Zabbix	mikoomi-mongodb	Monitors availability, resource utilization, health, performance and other important metrics.

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

Hosted (SaaS) Monitoring Tools

These are monitoring tools provided as a hosted service, usually on a subscription billing basis.

Name	Notes
Scout	Several plugins including: MongoDB Monitoring, MongoDB Slow Queries and MongoDB Replica Set Monitoring.
Server Density	Dashboard for MongoDB, MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.

11.3.2 Diagnosing Performance Issues

Degraded performance in MongoDB can be the result of an array of causes, and 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 lock 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's time to add additional capacity to the database.

Locks

MongoDB uses a locking system to ensure consistency; however, if certain operations are long-running, or a queue forms, performance slows as requests and operations wait for the lock. Because lock related slow downs can be intermittent, look to the data in the *globalLock* (page 640) section of the `serverStatus` response to asses if the lock has been a challenge to your performance. If `globalLock.currentQueue.total` (page 641) 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 might effect performance.

If `globalLock.totalTime` (page 640) is high in context of `uptime` (page 638) then the database has existed in a lock state for a significant amount of time. If `globalLock.ratio` (page 641) 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 173) and disk reads.

Memory Usage

Because MongoDB uses memory mapped files to store data, given a data set of sufficient size, the MongoDB process will allocate all memory available on the system for its use. Because of the way operating systems function, the amount of allocated RAM is not a useful reflection of MongoDB's state.

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. Consider *memory usage statuses* (page 642) to better understand MongoDB's memory utilization. Check the resident memory use (i.e. `mem.resident` (page 642):) if this exceeds the amount of system memory *and* there's a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

Also check the amount of mapped memory (i.e. `mem.mapped` (page 642).) 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 with deleterious effects on performance.

Page Faults

Page faults represent the number of times that 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` (page 643) value in the

serverStatus command. This data is only available on Linux systems.

Alone, page faults are minor and complete quickly; however, in aggregate, large numbers of page fault typically indicate that MongoDB is reading too much data from disk and can indicate a number of underlying causes and recommendations. 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 in high volume systems this also improves overall throughput.

If possible, 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 one or more *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 which can produce performance irregularities. Check the following fields in the *serverStatus* (page 637) document:

- globalLock.activeClients (page 641) 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:
  - connections.current (page 642) the total number of current clients that connect to the database instance.
  - connections.available (page 643) the total number of unused collections available for new clients.

Note: Unless limited by system-wide limits MongoDB has a hard connection limit of 20 thousand connections. You can modify system limits using the ulimit command, or by editing your system's http://docs.mongodb.org/manual/etc/sysctl file.

If requests are high because there are many 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 MongoDB drivers supported by 10gen 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 contains a database profiling system that can help identify inefficient queries and operations. Enable the profiler by setting the profile value using the following command in the mongo shell:

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

See Also:

The documentation of db.setProfilingLevel() for more information about this command.

Note: Because the database profiler can have an impact on the 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*.

The following profiling levels are available:

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

See the output of the profiler in the system.profile collection of your database. You can specify the slowms (page 627) to set a threshold above which the profiler considers operations "slow" and thus included in the level 1 profiling data. You may configure slowms (page 627) at runtime, as an argument to the db.setProfilingLevel() operation.

Additionally, mongod records all "slow" queries to its log (page 622), as defined by slowms (page 627). The data in system.profile does not persist between mongod restarts.

You can view the profiler's output by issuing the show profile command in the mongo shell, 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 (i.e. 100) is above the slowms (page 627) threshold.

See Also:

The Optimization wiki page addresses strategies that may improve the performance of your database queries and operations.

11.3.3 Replication and Monitoring

The primary administrative concern that requires monitoring with replica sets, beyond the requirements for any MongoDB instance is "replication lag." This refers to the amount of time that it takes a write operation on the *primary* to replicate to a *secondary*. Some very small delay period may be acceptable; however, as replication lag grows, two significant problems emerge:

- First, operations that have occurred in 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 the secondary will have to resync all data from the *primary* and rebuild all indexes. In normal circumstances this is uncommon given the typical size of the oplog, but it's an issue to be aware of.

For causes of replication lag, see *Replication Lag* (page 47).

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()
```

See the *Replica Set Status Reference* (page 659) document for a more in depth overview view of this output. In general watch the value of optimeDate. Pay particular attention to the difference in time between the *primary* and the *secondary* members.

The size of the operation log is only configurable during the first run using the *--oplogSize* (page 586) argument to the mongod command, or preferably the oplogSize (page 628) in the MongoDB configuration file. If you do

not specify this on the command line before running with the `--replSet` (page 586) option, `mongod` will create an default sized oplog.

By default the oplog is 5% of total available disk space on 64-bit systems.

See Also:

Change the Size of the Oplog (page 85)

11.3.4 Sharding and Monitoring

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

See Also:

See the Sharding wiki page for more information.

Config Servers

The *config database* provides a map of documents to shards. The cluster updates this map as *chunks* move between shards. When a configuration server becomes inaccessible, some sharding operations like moving chunks and starting `mongos` (page 676) instances become unavailable. However, clusters remain accessible from already-running `mongos` (page 676) instances.

Because inaccessible configuration servers can have a serious impact on the availability of a sharded cluster, you should monitor the configuration servers to ensure that the cluster remains well balanced and that `mongos` (page 676) instances can restart.

Balancing and Chunk Distribution

The most effective *sharded cluster* deployments require that *chunks* are evenly balanced between the shards. MongoDB has a background *balancer* process that distributes data such that chunks are always optimally distributed among the *shards*. Issue the `db.printShardingStatus()` or `sh.status()` command to the `mongos` (page 676) 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` (page 676) 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 might return a useful result set. The balancing process, which originates on a randomly selected `mongos` (page 676), 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.

11.4 Importing and Exporting MongoDB Data

Full *database instance backups* (page 180) are useful for disaster recovery protection and routine database backup operation; however, some cases require additional import and export functionality.

This document provides an overview of the import and export tools provided in distributions for MongoDB administrators. These utilities are useful when you want to backup or export a portion of your database without capturing the state of the entire database. 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.

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 backup 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 the "MongoDB Extended JSON" wiki page for more information about Use with care.

See Also:

See the "*Backup and Restoration Strategies* (page 180)" document for more information on backing up MongoDB instances. Additionally, consider the following references for commands addressed in this document:

- *mongoexport* (page 609)
- *mongorestore* (page 599)

11.4.1 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" wiki page.

11.4.2 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 and Restoration Strategies* (page 180)" 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.

11.4.3 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 *mongoimport* (page 606) and *mongoexport* (page 609) 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`* (page 610)",) `mongoexport` writes output to standard output (i.e. "stdout.") You can further narrow the results by supplying a query filter using the "*`--query`* (page 610)" and limit results to a single database using the "*`--db`* (page 610)" 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`* (page 610)" argument to return the export as a single *JSON* array. Use the "*`--csv`* (page 610)" file to return the result in CSV (comma separated values) format.

If your `mongod` instance is not running, you can use the "*`--dbpath`* (page 610)" 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`* (page 609)" and "*`--port`* (page 609)" 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 co
```

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`* (page 607)" option, `mongoimport` accepts input over standard input (e.g. "stdin.")

If you specify the "*`--upsert`* (page 608)" 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`* (page 607)" option to specify the location of your MongoDB instance's database files. Consider using the "*`--journal`* (page 607)" 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` (page 607)" 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.

11.5 Backup and Restoration Strategies

This document provides an inventory of database backup strategies for use with MongoDB. This document contains the following sections:

- *Backup Overview* (page 180) and *Production Considerations for Backup Strategies* (page 180) describe the approaches for backing up your MongoDB environment.
- *Using Block Level Backup Methods* (page 181) and *Using Binary Database Dumps for Backups* (page 185) describe specific strategies.
- *Sharded Cluster and Replica Set Considerations* (page 186) describes considerations specific to *replica sets* and *sharded clusters*.

11.5.1 Backup Overview

A robust backup strategy and well tested restoration process are crucial for production-grade deployments. The goal of a backup strategy is to produce a full and consistent copy of the data that you can use to bring up a new or replacement database instance.

With MongoDB, there are two major approaches to backups:

- Using system-level tools, like disk image snapshots.

  See *Using Block Level Backup Methods* (page 181).

- Using various capacities present in the `mongodump` tool.

  See *Using Binary Database Dumps for Backups* (page 185).

The methods described in this document operate by copying the data file on the disk level. If your system does not provide functionality for this kind of backup, see the section on *Using Binary Database Dumps for Backups* (page 185).

Ensuring that the state captured by the backup is consistent and usable is the primary challenge of producing backups of database systems. Backups that you cannot produce reliably, or restore from feasibly are worthless.

As you develop your backup system, take into consideration the specific features of your deployment, your use patterns, and your architecture.

Because every environment is unique it's important to regularly test the backups that you capture to ensure that your backup system is practically, and not just theoretically, functional.

11.5.2 Production Considerations for Backup Strategies

When evaluating a backup strategy for your node consider the following factors:

- Geography. Ensure that you move some backups away from the your primary database infrastructure. It's important to be able to restore your database if you lose access to a system or site.
- 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's important to consider the time of the backup schedule relative to peak usage and maintenance windows.

- System capabilities. In order to use some of the block-level snapshot tools requires special support on the operating-system or infrastructure level.
- Database configuration. *Replication* and *sharding* can affect the process, and impact of the backup implementation.
- 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.

With this information in hand you can begin to develop a backup plan for your database. Remember that all backup plans must be:

- Tested. If you cannot effectively restore your database from the backup, then your backups are useless. Test backup restoration regularly in practical situations to ensure that your backup system provides value.
- Automated. Database backups need to run regularly and automatically. Also automate tests of backup restoration.

11.5.3 Using Block Level Backup Methods

This section provides an overview of using disk/block level snapshots (i.e. *LVM* or storage appliance) to backup a MongoDB instance. These tools make a quick block-level backup of the device that holds MongoDB's data files. These methods complete quickly, work reliably, and typically provide the easiest backup systems method to implement.

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 to 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.

Backup With Journaling

If your system has snapshot capability and your `mongod` instance has journaling enabled, then you can use any kind of file system or volume/block level snapshot tool to create backups.

Warning: Changed in version 1.9.2. Journaling is only enabled by default on 64-bit builds of MongoDB.
To enable journaling on all other builds, specify `journal` (page 625) = `true` in the configuration or use the `--journal` (page 584) run-time option for `mongod`.

Many service providers provide a block-level backup service based on disk image snapshots. 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.

If you use Amazon's EBS service in a software RAID 10 configuration, use *LVM* to capture a consistent disk image. Also, see the special considerations described in *Amazon EBS in Software RAID 10 Configuration* (page 184).

The following sections provide 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.

Create 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-snap`. 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 you 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 Snapshots

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. Consider the following procedure to fully archive the data from the snapshot:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | tar -czf mdb-snap01.tar.gz
```

The above command sequence:

- Ensures that the `http://docs.mongodb.org/manual/dev/vg0/mdb-snap01` device is not mounted.
- Does a block level copy of the entire snapshot image using the `dd` command, and compresses the result in a gzipped tar archive in the current working directory.

  Warning: This command will create a large `tar.gz` file in your current working directory. Make sure that you run this command in a file system that has enough free space.

Restore Snapshot

To restore a backup created with the above method, use the following procedure:

```
lvcreate --size 1G --name mdb-new vg0
tar -xzf mdb-snap01.tar.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

The above sequence:

- 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.tar.gz` into the `mdb-new` disk image.
- Mounts the `mdb-new` disk image to the `http://docs.mongodb.org/manual/srv/mongodb` directory. Modify the mount point to correspond to your MongoDB data file location, or other location as needed.

Restore Directly from a Snapshot

To restore a backup without writing to a compressed `tar` archive, use the following sequence:

```
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 183) 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 tar -czf /opt/backup/mdb-snap01.tar.gz
lvcreate --size 1G --name mdb-new vg0
ssh username@example.com tar -xzf /opt/backup/mdb-snap01.tar.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Backup Without Journaling

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* that 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 Snapshot* (page 182).

3. To unlock the database after the snapshot has completed, use the following command in the `mongo` shell:

   ```
   db.fsyncUnlock();
   ```

   Note:

   ..versionchanged:: 2.0 In 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.

Amazon EBS in Software 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 a result you may:

- Flush all writes to disk and create a write lock to ensure consistent state during the backup process.

  If you choose this option see *Backup Without Journaling* (page 184).

- 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 Snapshot* (page 182).

11.5.4 Using Binary Database Dumps for Backups

This section 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.

See Also:

The *mongodump* (page 596) and *mongorestore* (page 599) documents contain complete documentation of these tools. If you have questions about these tools not covered here, please refer to these documents.

If your system has disk level snapshot capabilities, consider the backup methods described in *Using Block Level Backup Methods* (page 181).

Changed in version 2.2: In previous versions, `mongodump` would not store index definitions, nor would `mongorestore` rebuild indexes after restoring the data. In 2.2 `mongodump` will store index definitions *unless* you use the `--noIndexRestore` option. The corresponding `mongorestore` program will rebuild indexes when present in the dump.

Database Dump with `mongodump`

The `mongodump` utility can perform a live backup of data or can work against an inactive set of database files. The `mongodump` utility can create a dump for an entire server/database/collection (or part of a collection using of query), even when the database is running and active. If you run `mongodump` without any arguments, the command connects to the local database instance (e.g. `127.0.0.1` or `localhost`) and creates a database backup named `dump/` in the current directory.

Note: If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you can restore that dump only to a 2.2 database.

To limit the amount of data included in the database dump, you can specify `--database` and `--collection` (page 598) as options to the `mongodump` command. For example:

```
mongodump --collection collection --db test
```

This command creates a dump of the database in the `dump/` directory of the collection named `collection` in the database named `test`.

Use `--oplog` (page 598) option with `mongodump` to collect the *oplog* entries to build a point-in-time snapshot of the database. With `--oplog` (page 598), `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` (page 601), allows you to restore a backup that reflects a consistent and specific moment in time.

If your MongoDB instance is not running, you can use the `--dbpath` (page 597) 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:

```
mongodump --dbpath /srv/mongodb
```

Additionally, the `--host` (page 597) and `--port` (page 597) options allow you to specify a non-local host to connect to capture the dump. Consider the following example:

```
mongodump --host mongodb1.example.net --port 3017 --username user --password pass --out /opt/backup/m
```

On any `mongodump` command you may, as above, specify username and password credentials to specify database authentication.

Database Import with mongorestore

The mongorestore utility restores a binary backup created by mongodump. Consider the following example command:

```
mongorestore dump-2011-10-25/
```

Here, mongorestore imports the database backup located in the dump-2011-10-25 directory to the mongod instance running on the localhost interface. By default, mongorestore looks for a database dump in the dump/ directory and restores that. If you wish to restore to a non-default host, the *--host* (page 600) and *--port* (page 600) options allow you to specify a non-local host to connect to capture the dump. Consider the following example:

```
mongorestore --host mongodb1.example.net --port 3017 --username user --password pass /opt/backup/mong
```

On any mongorestore command you may specify username and password credentials, as above.

If you created your database dump using the *--oplog* (page 598) option to ensure a point-in-time snapshot, call mongorestore with the *--oplogReplay* (page 601) option, as in the following example:

```
mongorestore --oplogReplay
```

You may also consider using the *mongorestore --objcheck* (page 601) option to check the integrity of objects while inserting them into the database, or you may consider the *mongorestore --drop* (page 601) option to drop each collection from the database before restoring from backups. mongorestore also includes the ability to a 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.

```
mongorestore --dbpath /srv/mongodb --journal
```

Here, mongorestore restores the database dump located in dump/ folder into the data files located at http://docs.mongodb.org/manual/srv/mongodb. Additionally, the *--journal* (page 600) 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:

mongodump (page 596) and *mongorestore* (page 599).

11.5.5 Sharded Cluster and Replica Set Considerations

The underlying architecture of *sharded clusters* and *replica sets* presents several challenges for creating backups. This section describes how to make quality backups in environments with these configurations and how to perform restorations.

Back Up Sharded Clusters

Sharding complicates backup operations, because it is difficult to create a backup of a single moment in time from a distributed cluster of systems and processes.

Depending on the size of your data, you can back up the cluster as a whole or back up each mongod instance. The following section describes both procedures.

Back Up the Cluster as a Whole Using `mongodump`

If your *sharded cluster* comprises a small collection of data, you can connect to a `mongos` (page 676) and issue the `mongodump` command. You can use this approach if the following is true:

- It's possible to store the entire backup on one system or on a single storage device. Consider both backups of entire instances and incremental dumps of data.
- The state of the database at the beginning of the operation is not significantly different than the state of the database at the end of the backup. If the backup operation cannot capture a backup, this is not a viable option.
- The backup can run and complete without affecting the performance of the cluster.

Note: If you use `mongodump` without specifying the a database or collection, the output will contain both the collection data and the sharding config metadata from the *config servers* (page 110).

You cannot use the `--oplog` (page 598) option for `mongodump` when dumping from a `mongos` (page 676). This option is only available when running directly against a *replica set* member.

Back Up from All Database Instances

If your *sharded cluster* is to large for the `mongodump` command, then you must back up your data either by creating a snapshot of the cluster or by creating a binary dump of each database. This section describes both.

In both cases:

- The backups must capture the database in a consistent state.
- The sharded cluster must be consistent in itself.

This procedure describes both approaches:

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 126) 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 all shards so that your backups reflect your entire database system at a single point in time. Lock all shards 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 each shard `mongod` instance and block write operations.
   - Shutdown one of the *config servers* (page 110), to prevent all metadata changes during the backup process.

   Warning: If you do not lock all shards at the same time, the backup can reflect an inconsistent state that is impossible to restore from.

3. Use `mongodump` to backup one of the *config servers* (page 110). This backs up the cluster's metadata. You only need to back up one config server, as they all have replicas of the same information.

   Issue this command against one of the the config server itself or the `mongos` (page 676):

```
mongodump --db config
```

Note: In this situation `mongodump` will read from *secondary* nodes. See: *mongodump feature* (page 599) for more information.

4. Back up each shard. You may back up shards one at a time or in parallel. For each shard, do one of the following:
   - If your system has disk level snapshot capabilities, create a snapshot each shard. Use the procedures in *Using Block Level Backup Methods* (page 181).
   - Create a binary dump of each shard using the operations described in *Using Binary Database Dumps for Backups* (page 185).
5. Unlock all shards using the `db.fsyncUnlock()` method in the `mongo` shell.
6. Restore the balancer with the `sh.startBalancer()` method according to the *Disable the Balancer* (page 126) procedure.

   Use the following command sequence when connected to the `mongos` (page 676) with the `mongo` shell:

   ```
   use config
   sh.startBalancer()
   ```

Schedule Automated Backups

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 6:00 am and 11:00pm, server time. Schedule your backup operation to run *and complete* in this time. Ensure that the backup can complete during 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 Sharded Clusters

1. Stop all `mongod` and `mongos` (page 676) processes.
2. If shard hostnames have changed, you must manually update the `shards` collection in the *Config Database Contents* (page 675) to use the new hostnames. Do the following:
   (a) Start the three *config servers* (page 110) by issuing commands similar to the following, using values appropriate to your configuration:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27018
   ```

   (b) Restore the *Config Database Contents* (page 675) on each config server.
   (c) Start one `mongos` (page 676) instance.
   (d) Update the *Config Database Contents* (page 675) 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.
   - Data files for each *config server* (page 110), if you have not already done so in the previous step.

4. Restart the all the mongos (page 676) instances.
5. Restart all the mongod instances.
6. Connect to a mongos (page 676) instance from a mongo shell and run use the db.printShardingStatus() method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

Restore a Single Shard

Always restore *sharded clusters* as a whole. When you restore a single shard, keep in mind that the *balancer* process might have moved *chunks* onto or off of 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.
2. For all chunks that migrated away from this shard, you need not do anything. You do not need to delete these documents from the shard because the chunks are automatically filtered out from queries by mongos (page 676).
3. For chunks that migrated to this shard since the last backup, you must manually recover the chunks. To determine what chunks have moved, view the changelog collection in the *Config Database Contents* (page 675).

Replica Sets

In most cases, backing up data stored in a *replica set* is similar to backing up data stored in a single instance. It's possible to lock a single *secondary* or *slave* database and then create a backup from that instance. When you unlock the database, the secondary or slave will catch up with the *primary* or *master*. You may also chose to deploy a dedicated *hidden member* for backup purposes.

If you have a *sharded cluster* where each *shard* is itself a replica set, you can use this method 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/non-master node to create backups is particularly advantageous in that the backup operation does not affect the performance of the primary or master. 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.

11.6 Linux ulimit Settings

The Linux kernel provides a system to limit and control the number of threads, connections, and open files on a per-process and per-user basis. These limits prevent single users from using too many system resources. Sometimes, these limits, as configured by the distribution developers, are too low for MongoDB and can cause a number of issues in the course of normal MongoDB operation. Generally, MongoDB should be the only user process on a system, to prevent resource contention.

11.6.1 Resource Utilization

mongod and mongos (page 676) 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 (page 676) instances, like other processes:

- 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 (page 625) 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 296), 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 (page 676) must maintain connects to all config servers and all shards, which includes all members of all replica sets.

For mongos (page 676), consider the following behaviors:

- mongos (page 676) instances maintain a connection pool to each shard so that the mongos (page 676) can reuse connections and quickly fulfill requests without needing to create new connections.
- You can limit the number of incoming connections using the maxConns (page 622) run-time option:

  ```
  :option:`--maxConns <mongos --maxConns>`
  ```

  By restricting the number of incoming connections you can prevent a cascade effect where the mongos (page 676) creates too many connections on the mongod instances.

  Note: You cannot set maxConns (page 622) to a value higher than *20000*.

11.6.2 Review and Set Resource Limits

ulimit

You can use the ulimit command at the system prompt to check system limits, as in the following example:

```
$ ulimit -a
-t: cpu time (seconds)         unlimited
-f: file size (blocks)         unlimited
-d: data seg size (kbytes)     unlimited
-s: stack size (kbytes)        8192
-c: core file size (blocks)    0
-m: resident set size (kbytes) unlimited
-u: processes                  192276
-n: file descriptors           21000
-l: locked-in-memory size (kb) 40000
-v: address space (kb)         unlimited
-x: file locks                 unlimited
-i: pending signals            192276
-q: bytes in POSIX msg queues  819200
```

```
-e: max nice                    30
-r: max rt priority             65
-N 15:                          unlimited
```

`ulimit` refers to the per-*user* limitations for various resources. Therefore, if your `mongod` instance executes as a user that is also running multiple processes, or multiple `mongod` processes, you might see contention for these resources. Also, be aware that the `processes` value (i.e. `-u`) refers to the combined number of distinct processes and sub-process threads.

You can change `ulimit` settings by issuing a command in the following form:

```
ulimit -n <value>
```

Substitute the `-n` option for any possible value in the output of `ulimit -a`.

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/manual/proc` file system to see the current limitations on a running process.

Depending on your system's configuration, and default settings, any change to system limits made using `ulimit` may revert following system a system restart. Check your distribution and operating system documentation for more information.

`/proc` File System

Note: This section applies only to Linux operating systems.

The `http://docs.mongodb.org/manual/proc` file-system stores the per-process limits in the file system object located at `http://docs.mongodb.org/manual/proc/<pid>/limits`, where `<pid>` is the process's *PID* or process identifier. You can use the following `bash` function to return the content of the `limits` object for a process or processes with a given name:

```
return-limits(){

     for process in $@; do
          process_pids=`ps -C $process -o pid --no-headers | cut -d " " -f 2`

          if [ -z $@ ]; then
             echo "[no $process running]"
          else
             for pid in $process_pids; do
                   echo "[$process #$pid -- limits]"
                   cat /proc/$pid/limits
             done
          fi

     done

}
```

You can copy and paste this function into a current shell session or load it as part of a script. Call the function with one the following invocations:

```
return-limits mongod
return-limits mongos
return-limits mongod mongos
```

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
Max realtime priority     65                   65
Max realtime timeout      unlimited            unlimited            us
```

11.6.3 Recommended Settings

Every deployment may have unique requirements and settings; however, the following thresholds and settings are particularly important for `mongod` and `mongos` (page 676) deployments:

- `-f` (file size): `unlimited`
- `-t` (cpu time): `unlimited`
- `-v` (virtual memory): `unlimited`
- `-n` (open files): `64000`
- `-m` (memory size): `unlimited`
- `-u` (processes/threads): `32000`

Always remember to restart your `mongod` and `mongos` (page 676) instances after changing the `ulimit` settings to make sure that the settings change takes effect.

See Also:

- *Replica Set Administration* (page 38)
- *Replication Architectures* (page 51)
- *Sharded Cluster Administration* (page 113)
- *Sharded Cluster Architectures* (page 131)
- *indexes* (page 231)
- *Indexing Operations* (page 238)

CHAPTER

TWELVE

TUTORIALS

The following tutorials describe basic administrative procedures for MongoDB deployments:

12.1 Recover MongoDB Data following Unexpected Shutdown

If MongoDB does not shutdown cleanly [1] the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption.

To prevent data inconsistency and corruption, always shut down the database cleanly and use the *durability journaling* (page 625). The journal writes data to disk every 100 milliseconds by default and ensures that MongoDB can 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` (page 624) and allow MongoDB to resync the data.

See Also:

The *Administration* (page 161) documents, including *Syncing* (page 64), and the documentation on the `repair` (page 626), `repairpath` (page 627), and `journal` (page 625) settings.

12.1.1 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 *synchronize* (page 64) your data.

If the `mongod.lock` file in the data directory specified by `dbpath` (page 624), `http://docs.mongodb.org/manual/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 remove the lockfile and run repair. If you run repair when the `mongodb.lock` file exists without the *`mongod --repairpath`* (page 585) option, you will see a message that contains the following line:

[1] To ensure a clean shut down, use the *`mongod --shutdown`* (page 586) option, your control script, "Control-C" (when running `mongod` in interactive mode,) or `kill $(pidof mongod)` or `kill -2 $(pidof mongod)`.

```
old lock file: /data/db/mongod.lock. probably means unclean shutdown
```

You must remove the lockfile **and** run the repair operation before starting the database normally using 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 180) or resync from an intact member of the set, as described in *Resyncing a Member of a Replica Set* (page 45).

There are two processes to repair data files that result from an unexpected shutdown:

1. Use the `--repair` (page 585) option in conjunction with the `--repairpath` (page 585) 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` (page 585) 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.

Procedures

To repair your data files using the `--repairpath` (page 585) option to preserve the original data files unmodified:

1. Start `mongod` using `--repair` (page 585) 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 `http://docs.mongodb.org/manual/data/db0` directory.

2. Start `mongod` using the following invocation to point the `dbpath` (page 624) at `http://docs.mongodb.org/manual/data/db2`:

   ```
   mongod --dbpath /data/db0
   ```

   Once you confirm that the data files are operational you may delete or archive the data files in the `http://docs.mongodb.org/manual/data/db` directory.

To repair your data files without preserving the original files, do not use the `--repairpath` (page 585) option, as in the following procedure:

1. Remove the stale lock file:

   ```
   rm /data/db/mongod.lock
   ```

   Replace `http://docs.mongodb.org/manual/data/db` with your `dbpath` (page 624) where your MongoDB instance's data files reside.

   Warning: After you remove the `mongod.lock` file you *must* run the `--repair` (page 585) process before using your database.

2. Start `mongod` using *`--repair`* (page 585) 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 `http://docs.mongodb.org/manual/data/db` directory.

3. Start `mongod` using the following invocation to point the `dbpath` (page 624) at `http://docs.mongodb.org/manual/data/db`:

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

12.1.2 `mongod.lock`

In normal operation, you should **never** remove the `mongod.lock` file and start `mongod`. Instead use 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 180) or if running as a *replica set* resync from an intact member of the set, as described in *Resyncing a Member of a Replica Set* (page 45).

12.2 Convert a Replica Set to a Replicated Sharded Cluster

12.2.1 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.

12.2.2 Process

Install MongoDB according to the instructions in the *MongoDB Installation Tutorial* (page 9).

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 197).

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:

   - `http://docs.mongodb.org/manual/data/example/firstset1`
   - `http://docs.mongodb.org/manual/data/example/firstset2`
   - `http://docs.mongodb.org/manual/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* (page 586) option restricts the size of the operation log (i.e. oplog) for each `mongod` instance to 700MB. Without the *--oplogSize* (page 586) 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":
                               }
```

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:

   - `http://docs.mongodb.org/manual/data/example/config1`
   - `http://docs.mongodb.org/manual/data/example/config2`
   - `http://docs.mongodb.org/manual/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` (page 676) 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` (page 590) (1MB works well.) The default `chunkSize` (page 630) 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` (page 630) options specify the *configuration databases* (e.g. `localhost:20001`, `localhost:20002`, and `localhost:2003`). The `mongos` (page 676) instance runs on the default "Mon-

goDB" 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* (page 589) option, as 27017 is the default port.

4. Add the first shard in mongos (page 676). In a new terminal window or GNU Screen session, add the first shard, according to the following procedure:

   (a) Connect to the mongos (page 676) 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:
   - http://docs.mongodb.org/manual/data/example/secondset1
   - http://docs.mongodb.org/manual/data/example/secondset2
   - http://docs.mongodb.org/manual/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 (page 628) 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 (page 676) 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 109) and *Shard Key* (page 133) sections.

Shard the Collection

Issue the following command:

```
use admin
db.runCommand( { shardCollection : "test.test_collection", key : {"number":1} })
{ "collectionsharded" : "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` (page 676) 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.

12.3 Copy Databases Between Instances

12.3.1 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 `mongod` shell.

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 and Restoration Strategies* (page 180) and *Importing and Exporting MongoDB Data* (page 177) 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.

12.3.2 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` (page 676).
- 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 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 on the output of `db.stats()` see *Database Statistics Reference* (page 651) document.

12.3.3 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" )
```

Consider the following tutorials located in other sections of this Manual.

12.4 Installation

- *Install MongoDB on Linux* (page 17)
- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 9)
- *Install MongoDB on Debian* (page 15)
- *Install MongoDB on Ubuntu* (page 12)
- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 22)

12.5 Replica Sets

- *Deploy a Replica Set* (page 73)
- *Convert a Standalone to a Replica Set* (page 76)
- *Add Members to a Replica Set* (page 77)
- *Deploy a Geographically Distributed Replica Set* (page 79)

- *Change the Size of the Oplog* (page 85)
- *Force a Member to Become Primary* (page 87)
- *Change Hostnames in a Replica Set* (page 89)
- *Convert a Secondary to an Arbiter* (page 93)
- *Reconfigure a Replica Set with Unavailable Members* (page 95)

12.6 Sharding

- *Deploy a Sharded Cluster* (page 141)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 195)
- *Add Shards to an Existing Cluster* (page 144)
- *Remove Shards from an Existing Sharded Cluster* (page 145)

12.7 Basic Operations

- *Recover MongoDB Data following Unexpected Shutdown* (page 193)
- *Copy Databases Between Instances* (page 201)
- *Expire Data from Collections by Setting TTL* (page 296)

12.8 Security

- *Configure Linux iptables Firewall for MongoDB* (page 215)
- *Configure Windows netsh Firewall for MongoDB* (page 219)
- *Control Access to MongoDB Instances with Authentication* (page 223)

Part VI

Security

CHAPTER

THIRTEEN

SECURITY PRACTICES AND MANAGEMENT

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. This document 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 *Defense In Depth* approaches 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. See the *Strategies for Reducing Risk* (page 207) section for more information.

13.1 Strategies for Reducing Risk

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` (page 676) and application instances) in a *trusted environment*. Trusted environments use the following strategies to control access:

- network filter (e.g. firewall) rules that block all connections from unknown systems to MongoDB components.
- bind `mongod` and `mongos` (page 676) instances to specific IP addresses to limit accessibility.
- limit MongoDB programs to non-public local networks, and virtual private networks.

You may further reduce risk by:

- requiring authentication for access to MongoDB instances.
- requiring strong, complex, single purpose authentication credentials. This should be part of your internal security policy but is not currently configurable in MongoDB.
- deploying a model of least privilege, where all users have *only* the amount of access they need to accomplish required tasks, and no more.
- following the best application development and deployment practices, which includes: validating all inputs, managing sessions, and application-level access control.

Continue reading this document for more information on specific strategies and configurations to help reduce the risk exposure of your application.

13.2 Vulnerability Notification

10gen takes the security of MongoDB and associated products very seriously. If you discover a vulnerability in MongoDB or another 10gen product, or would like to know more about our vulnerability reporting and response process, see the *Vulnerability Notification* (page 213) document.

13.3 Networking Risk Exposure

13.3.1 Interfaces and Port Numbers

The following list includes all default ports used by MongoDB:

By default, listens for connections on the following ports:

27017 This is the default port `mongod` and `mongos` (page 676) instances. You can change this port with `port` (page 622) or `--port` (page 582).

27018 This is the default port when running with `--shardsvr` (page 588) runtime operation or `shardsvr` (page 629) setting.

27019 This is the default port when running with `--configsvr` (page 587) runtime operation or `configsvr` (page 629) setting.

28017 This is the default port for the web status page. This is always accessible at a port that is `1000` greater than the port determined by `port` (page 622).

By default MongoDB programs (i.e. `mongos` (page 676) and `mongod`) will bind to all available network interfaces (i.e. IP addresses) on a system. The next section outlines various runtime options that allow you to limit access to MongoDB programs.

13.3.2 Network Interface Limitation

You can limit the network exposure with the following configuration options:

- the `nohttpinterface` (page 625) setting for `mongod` and `mongos` (page 676) 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`* (page 584) or *`mongos --nohttpinterface`*. Authentication does not control or affect access to this interface.

  Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

- the `port` (page 622) setting for `mongod` and `mongos` (page 676) instances.

  Changes the main port on which the `mongod` or `mongos` (page 676) instance listens for connections. Changing the port does not menacingly reduce risk or limit exposure.

  You may also specify this option on the command line as *`mongod --port`* (page 582) or *`mongos --port`* (page 589).

  Whatever port you attach `mongod` and `mongos` (page 676) instances to, you should only allow trusted clients to connect to this port.

- the `rest` (page 626) setting for `mongod` and `mongos` (page 676) instances.

  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`* (page 585).

  Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

- the `bind_ip` (page 622) setting for `mongod` and `mongos` (page 676) 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` (page 622) a comma separated list of IP addresses. You can use the *`mongod --bind_ip`* (page 582) and *`mongos --bind_ip`* (page 589) 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` (page 676) 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.

13.3.3 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.
- 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 consider the following documents:

- *Configure Linux iptables Firewall for MongoDB* (page 215)
- *Configure Windows netsh Firewall for MongoDB* (page 219)

For best results and to minimize overall exposure, ensure that *only* traffic from trusted sources can reach `mongod` and `mongos` (page 676) instances and that the `mongod` and `mongos` (page 676) instances can only connect to trusted outputs.

See Also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2 wiki page, which addresses Amazon's Security Groups and other EC2-specific security features.

13.3.4 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, using a VPN connection to control access to your MongoDB instance, you can prevent tampering and "man-in-the-middle" attacks.

13.4 Operations

Always run the `mongod` or `mongos` (page 676) 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` (page 676) 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.

You can disable anonymous access to the database by enabling authentication using the `auth` (page 624) as detailed in the *Authentication* (page 210) section.

13.5 Authentication

MongoDB provides basic support for authentication with the `auth` (page 624) setting. For multi-instance deployments (i.e. *replica sets*, and *sharded clusters*) use the `keyFile` (page 623) setting, which implies `auth` (page 624), and allows intra-deployment authentication and operation. Be aware of the following behaviors of MongoDB's authentication system:

- Authentication is **disabled** by default.
- MongoDB provisions access on a per-database level. Users either have *read only* access to a database or *normal* access to a database that permits full read and write access to the database. *Normal* access conveys the ability to add additional users to the database.
- The `system.users` collection in each database stores all credentials. You can query the authorized users with the following operation:

  ```
  db.system.users.find()
  ```

- The `admin` database is unique. Users with *normal* access to the `admin` database have read and write access to all databases. Users with *read only* access to the `admin` database have read only access to all databases.

  Additionally the `admin` database exposes several commands and functionality, such as `listDatabases`.
- Once authenticated a *normal* user has full read and write access to a database.
- If you have authenticated to a database as a normal, read and write, user; authenticating as a read-only user on the same database will invalidate the earlier authentication, leaving the current connection with read only access.
- If you have authenticated to the `admin` database as normal, read and write, user; logging into a *different* database as a read only user will *not* invalidate the authentication to the `admin` database. In this situation, this client will be able to read and write data to this second database.
- When setting up authentication for the first time you must either:
  1. add at least one user to the `admin` database before starting the `mongod` instance with `auth` (page 624).
  2. add the first user to the `admin` database when connected to the `mongod` instance from a `localhost` connection. [1]

[1] Because of SERVER-6591, you cannot add the first user to a sharded cluster using the `localhost` connection in 2.2. 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` (page 623).

New in version 2.0: Support for authentication with sharded clusters. Before 2.0 sharded clusters *had* to run with trusted applications and a trusted networking configuration. Consider the *Control Access to MongoDB Instances with Authentication* (page 223) document which outlines procedures for configuring and maintaining users and access with MongoDB's authentication system.

13.6 Interfaces

Simply limiting access to a `mongod` is not sufficient for totally controlling risk exposure. Consider the recommendations in the following section, for limiting exposure other interface-related risks.

13.6.1 JavaScript and the Security of the `mongo` Shell

Be aware of the following capabilities and behaviors of the `mongo` shell:

- `mongo` will evaluate a `.js` file passed to the *`mongo --eval`* (page 592) option. The `mongo` shell does not validate the input of JavaScript input to *`--eval`* (page 592).
- `mongo` will evaluate a `.mongorc.js` file before starting. You can disable this behavior by passing the *`mongo --norc`* ' option.

  On Linux and Unix systems, `mongo` reads the `.mongorc.js` file from *`$HOME`*`/.mongorc.js` (i.e. `~/.mongorc.js`), and Windows `mongo.exe` reads the `.mongorc.js` file from *`%HOME%`*`.mongorc.js` or *`%HOMEDRIVE%%HOMEPATH%`*`.mongorc.js`.

13.6.2 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` (page 676) instance. The HTTP interface is always available on the the port numbered `1000` greater than the primary `mongod` port. By default this is `28017`, but is indirectly set using the `port` (page 622) option which allows you to configure the primary `mongod` port.

Without the `rest` (page 626) 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` (page 625) run time option or the *`--nohttpinterface`* (page 584) command line option.

13.6.3 REST API

The REST API to MongoDB provides additional information and write access on top of the HTTP Status interface. The REST interface is *disabled* by default, and is not recommended for production use.

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.

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 if when running with `auth` (page 624) enabled.

See the following documents for instructions on restricting access to the REST API interface:

- *Configure Linux iptables Firewall for MongoDB* (page 215)
- *Configure Windows netsh Firewall for MongoDB* (page 219)

13.7 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, 10gen has a partnership with Gazzang 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 and can access this data. The Gazzang software ensures that the cryptographic keys remain safe and ensures compliance with standards including HIPPA, PCI-DSS, and FERPA. For more information consider the following resources:

- Datasheet
- Webinar

CHAPTER

FOURTEEN

VULNERABILITY NOTIFICATION

10gen values the privacy and security of all users of MongoDB, and we work very hard to ensure that MongoDB and related tools minimize risk exposure and increase the security and integrity of data and environments using MongoDB.

14.1 Notification

If you believe you have discovered a vulnerability in MongoDB or a related product or have experienced a security incident related to MongoDB, please report these issues so that 10gen can respond appropriately and work to prevent additional issues in the future. All vulnerability reports should contain as much information as possible so that we 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.

10gen will respond to all vulnerability notifications within 48 hours.

14.1.1 Jira

10gen prefers jira.mongodb.org for all communication regarding MongoDB and related products.

Submit a ticket in the "Core Server Security" project, at: <https://jira.mongodb.org/SECURITY/>. The ticket number will become reference identification for the issue for the lifetime of the issue, and you can use this identifier for tracking purposes.

10gen will respond to any vulnerability notification received in a Jira case posted to the SECURITY project.

14.1.2 Email

While Jira is the preferred communication vector, you may also report vulnerabilities via email to <security@10gen.com>.

You may encrypt email using our public key, to ensure the privacy of a any sensitive information in your vulnerability report.

10gen will respond to any vulnerability notification received via email with email which will contain a reference number (i.e. a ticket from the SECURITY project,) Jira case posted to the SECURITY project.

14.1.3 Evaluation

10gen will validate all submitted vulnerabilities. 10gen will use Jira to track all communications regarding the vulnerability, which may include requests for clarification and for additional information. If needed 10gen representatives can set up a conference call to exchange information regaining the vulnerability.

14.1.4 Disclosure

10gen requests that you do *not* publicly disclose any information regarding the vulnerability or exploit until 10gen has had the opportunity to analyze the vulnerability, respond to the notification, and to notify key users, customers, and partners if needed.

The amount of time required to validate a reported vulnerability depends on the complexity and severity of the issue. 10gen takes all required vulnerabilities very seriously, and will always ensure that there is a clear and open channel of communication with the reporter of the vulnerability.

After validating the issue, 10gen will coordinate 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.

CHAPTER

FIFTEEN

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 over view of security practices and risk management for MongoDB, see *Security Practices and Management* (page 207).

See Also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2 wiki page, which addresses Amazon's Security Groups and other EC2-specific security features.

15.1 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 only the following two chains:

`INPUT` Controls all incoming traffic.

`OUTPUT` Controls all outgoing traffic.

Given the *default ports* (page 208) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod` and `mongos` (page 676) 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 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 218).

15.2 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` (page 622) configuration setting, you will need to modify the rules accordingly.

15.2.1 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 -
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/24 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
```

15.2.2 Traffic to and from mongos Instances

mongos (page 676) instances provide query routing for *sharded clusters*. Clients connect to mongos (page 676) instances, which behave from the client's perspective as mongod instances. In turn, the mongos (page 676) 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 216) section as an example.

15.2.3 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* (page 587) option. [1] 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 -
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 (page 676) 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 -
```

Replace <ip-address> with the address address of the mongos (page 676) instances and the shard mongod instances.

[1] You can also run a config server by setting the configsvr (page 629) option in a configuration file.

15.2.4 Traffic to and from a MongoDB Shard Server

For shard servers, running as *mongod --shardsvr* (page 588) [2] Because the default port number when running with shardsvr (page 629) 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 -
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. [3]

Furthermore, shards need to be able make outgoing connections to:

- all mongos (page 676) 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 (page 676) instances:

```
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

15.2.5 Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the *--discover* (page 612) needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos (page 676) 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,ESTABLISH
   ```

   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 (page 629), the rule would resemble the following:

   ```
   iptables -A INPUT -s <ip-address> -p tcp --destination-port 28018 -m state --state NEW,ESTABLISH
   ```

   For config server mongod instances running with configsvr (page 629), the rule would resemble the following:

   ```
   iptables -A INPUT -s <ip-address> -p tcp --destination-port 28019 -m state --state NEW,ESTABLISH
   ```

   ---

[2] You can also specify the shard server option using the shardsvr (page 629) setting in the configuration file. Shard members are also often conventional replica sets using the default port.

[3] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

15.3 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
```

15.4 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:

15.4.1 Make all iptables Rules Persistent

By default all iptables rules are 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 http://docs.mongodb.org/manual/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 http://docs.mongodb.org/manual/etc/network/if-up.d/ipta file with other similar operations.q

15.4.2 List all iptables Rules

To list all of currently applied iptables rules, use the following operation at the system shell.

```
iptables --L
```

15.4.3 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 218) section.

CHAPTER

SIXTEEN

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 over view of security practices and risk management for MongoDB, see *Security Practices and Management* (page 207).

See Also:

Windows Firewall documentation from Microsoft.

16.1 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 208) 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 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.

16.2 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` (page 622) configuration setting, you will need to modify the rules accordingly.

16.2.1 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 lo
```

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\b
```

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\b
```

16.2.2 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=
```

16.2.3 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`* (page 587) option. [1] 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 prot
```

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 protoco
```

Replace `<ip-address>` with the addresses of the `mongos.exe` instances and the shard `mongod.exe` instances.

[1] You can also run a config server by setting the `configsvr` (page 629) option in a configuration file.

16.2.4 Traffic to and from a MongoDB Shard Server

For shard servers, running as *mongod --shardsvr* (page 588) [2] Because the default port number when running with shardsvr (page 629) 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=
netsh advfirewall firewall add rule name="Open mongod shardsvr outbound" dir=out action=allow protoco
```

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.
- all mongod.exe instances in other shards. [3]

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
```

16.2.5 Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the *--discover* (page 612) 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=all
   ```

   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 (page 629), the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongos HTTP monitoring inbound" dir=in action=all
   ```

   For config server mongod.exe instances running with configsvr (page 629), the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongod configsvr HTTP monitoring inbound" dir=in
   ```

   ---

[2] You can also specify the shard server option using the shardsvr (page 629) setting in the configuration file. Shard members are also often conventional replica sets using the default port.

[3] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

16.3 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 is accessible from `netsh`.

16.3.1 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=270
```

16.3.2 List All *Windows Firewall* Rules

To return a list of all *Windows Firewall* rules:

```
netsh advfirewall firewall show rule name=all
```

16.3.3 Reset *Windows Firewall*

To reset the *Windows Firewall* rules:

```
netsh advfirewall reset
```

16.3.4 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"
```

CHAPTER

SEVENTEEN

CONTROL ACCESS TO MONGODB INSTANCES WITH AUTHENTICATION

MongoDB provides a basic authentication system, that you can enable with the `auth` (page 624) and `keyFile` (page 623) configuration settings. [1] See the *authentication* (page 210) section of the *Security Practices and Management* (page 207) document.

This document contains an overview of all operations related to authentication and managing a MongoDB deployment with authentication.

See Also:

The *Security Considerations* (page 164) section of the *Run-time Database Configuration* (page 163) document for more information on configuring authentication.

17.1 Adding Users

When setting up authentication for the first time you must either:

1. add at least one user to the `admin` database before starting the `mongod` instance with `auth` (page 624).
2. add the first user to the `admin` database when connected to the `mongod` instance from a `localhost` connection. [2]

Begin by setting up the first administrative user for the `mongod` instance.

17.1.1 Add an Administrative User

About administrative users

Administrative users are those users that have "normal" or read and write access to the `admin` database.

If this is the first administrative user, [#auth-disabled-maintenance] connect to the `mongod` on the `localhost` interface using the `mongo` shell. Then, issue the following command sequence to switch to the `admin` database context and add the administrative user:

[1] Use the *`--auth`* (page 583) *`--keyFile`* (page 583) options on the command line.

[2] Because of SERVER-6591, you cannot add the first user to a sharded cluster using the `localhost` connection in 2.2. 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` (page 623).

```
use admin
db.addUser("<username>", "<password>")
```

Replace <username> and <password> with the credentials for this administrative user.

17.1.2 Add a Normal User to a Database

To add a user with read and write access to a specific database, in this example the records database, connect to the mongod instance using the mongo shell, and issue the following sequence of operations:

```
use records
db.addUser("<username>", "<password>")
```

Replace <username> and <password> with the credentials for this user.

17.1.3 Add a Read Only User to a Database

To add a user with read only access to a specific database, in this example the records database, connect to the mongod instance using the mongo shell, and issue the following sequence of operations:

```
use records
db.addUser("<username>", "<password>", true)
```

Replace <username> and <password> with the credentials for this user.

17.2 Administrative Access in MongoDB

Although administrative accounts have access to all databases, these users must authenticate against the admin database before changing contexts to a second database, as in the following example:

Example

Given the superAdmin user with the password Password123, and access to the admin database.

The following operation in the mongo shell will succeed:

```
use admin
db.auth("superAdmin", "Password123")
```

However, the following operation will fail:

```
use test
db.auth("superAdmin", "Password123")
```

Note: If you have authenticated to the admin database as normal, read and write, user; logging into a *different* database as a read only user will *not* invalidate the authentication to the admin database. In this situation, this client will be able to read and write data to this second database.

17.3 Authentication on Localhost

The behavior of `mongod` running with `auth` (page 624), when connecting from a client over the localhost interface (i.e. a client running on the same system as the `mongod`,) varies slightly between before and after version 2.2.

In general if there are no users for the `admin` database, you may connect via the localhost interface. For sharded clusters running version 2.2, if `mongod` is running with `auth` (page 624) then all users connecting over the localhost interface must authenticate, even if there aren't any users in the `admin` database.

17.4 Password Hashing Insecurity

In version 2.2 and earlier:

- the *normal* users of a database all have access to the `system.users` collection, which contains the user names and a hash of all user's passwords. [#read-and-write-system-users]
- if a user has the same password in multiple databases, the hash will be the same on all database. A malicious user could exploit this to gain access on a second database use a different users' credentials.

As a result, always use unique username and password combinations on for each database.

Thanks to Will Urbanski, from Dell SecureWorks, for identifying this issue.

17.5 Configuration Considerations for Authentication

The following sections, outline practices for enabling and managing authentication with specific MongoDB deployments:

- *Security Considerations for Replica Sets* (page 46)
- *Security Considerations for Sharded Clusters* (page 113)

17.6 Generate a Key File

Use the following command at the system shell to generate pseudo-random content for a key file:

```
openssl rand -base64 753
```

Note: Be aware that MongoDB strips whitespace characters (e.g. `x0d`, `x09`, and `x20`,) for cross-platform convenience. As a result, the following keys are identical:

```
$ 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
```

Part VII

Indexes

Indexes provide high performance read operations for frequently used queries. Indexes are particularly useful where the total size of the documents exceeds the amount of available RAM.

For basic concepts and options, see *Indexing Overview* (page 231). For procedures and operational concerns, see *Indexing Operations* (page 238). For information on how applications might use indexes, see *Indexing Strategies* (page 242).

CHAPTER

EIGHTEEN

DOCUMENTATION

The following is the outline of the main documentation:

18.1 Indexing Overview

This document provides an overview of indexes in MongoDB, including index types and creation options. For operational guidelines and procedures, see the *Indexing Operations* (page 238) document. For strategies and practical approaches, see the *Indexing Strategies* (page 242) document.

18.1.1 Synopsis

An index is a data structure that allows you to quickly locate documents based on the values stored in certain specified fields. Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB supports indexes on any field or sub-field contained in documents within a MongoDB collection. MongoDB indexes have the following core features:

- MongoDB defines indexes on a per-*collection* level.
- Indexes often dramatically increase the performance of queries; however, each index creates a slight overhead for every write operation.
- Every query, including update operations, use one and only one index. The query optimizer selects the index empirically by occasionally running alternate query plans and by selecting the plan with the best response time for each query type. You can override the query optimizer using the `cursor.hint()` method.
- You can create indexes on a single field or on multiple fields using a *compound index* (page 233).
- When the index covers queries, the database returns results more quickly than queries that have to scan many individual documents. An index "covers" a query if the keys of the index stores all the data that the query must return. See *Use Compound Indexes to Support Several Different Queries* (page 243) for more information.
- Using queries with good index coverage will reduce the number of full documents that MongoDB needs to store in memory, thus maximizing database performance and throughput.

18.1.2 Index Types

All indexes in MongoDB are "B-Tree" indexes. In the `mongo` shell, the helper `ensureIndex()` provides a method for creating indexes. This section provides an overview of the types of indexes available in MongoDB as well as an introduction to their use.

_id

The _id index is a *unique index* (page 234) [1] on the _id field, and MongoDB creates this index by default on all collections. [2] You cannot delete the index on _id.

The _id field is the *primary key* for the collection, and every document *must* have a unique _id field. You may store any unique value in the _id field. The default value of _id is *ObjectID* on every insert() <db.collection.insert()‘ operation. An *ObjectId* is a 12-byte unique identifiers 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*.

Secondary Indexes

All indexes in MongoDB are *secondary indexes*. You can create indexes on any field within any document or sub-document. Additionally, you can create compound indexes with multiple fields, so that a single query can match multiple components using the index without needing to scan (as many) actual documents.

In general, you should have secondary indexes that support all of your primary, common, and user-facing queries and require MongoDB to scan the fewest number of documents possible.

To create a secondary index, use the ensureIndex() method. The argument to ensureIndex() will resemble the following in the MongoDB shell:

```
{ "field": 1 }
{ "field0.field1": 1 }
{ "field0": 1, "field1": 1 }
```

For each field in the index you will specify either 1 for an ascending order or -1 for a descending order, which represents the order of the keys in the index. For indexes with more than one key (i.e. “compound indexes,”) the sequence of fields is important.

Embedded Fields

You can create indexes on fields that exist in sub-documents within your collection. Consider the collection 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 } )
```

Introspecting sub-documents in this way is commonly called “dot notation.”

[1] Although the index on _id *is* unique, the getIndexes() method will *not* print unique: true in the mongo shell.

[2] Before version 2.2 capped collections did not have an _id field. In 2.2, all capped collections have an _id field, except those in the local *database*. See the *release notes* (page 698) for more information.

Compound Indexes

MongoDB supports "compound indexes," where a single index structure holds references to multiple fields within a collection's documents. Consider the collection `products` that holds documents that resemble the following example document:

```
{
 "_id": ObjectId(...)
 "item": "Banana"
 "category": ["food", "produce", "grocery"]
 "stock": 4
 "type": cases
 "arrival": Date(...)
}
```

Most queries probably select on the `item` field, but a significant number of queries will also check the `stock` field. You can specify a single compound index to support both of these queries:

```
db.products.ensureIndex( { "item": 1, "stock": 1 } )
```

MongoDB will be able to use this index to support queries that select the `item` field as well as those queries that select the `item` field **and** the `stock` field. However, this index will not be useful for queries that select *only* the `stock` field.

Note: The order of fields in a compound index is very important. In the previous example, the index will contain references to documents sorted by the values of the `item` field, and within each item, sorted by values of the `stock` field.

Ascending and Descending

Indexes store references to fields in either ascending or descending order. For single-field indexes, the order of keys doesn't matter, because MongoDB can traverse the index in either direction. However, for compound indexes, if you need to order results against two fields, sometimes you need the index fields running in opposite order relative to each other.

To specify an index with a descending order, use the following form:

```
db.products.ensureIndex( { "field": -1 } )
```

More typically in the context of a *compound index* (page 233), the specification would resemble the following prototype:

```
db.products.ensureIndex( { "field0": 1, "field1": -1 } )
```

Consider a collection of event data that includes both usernames and a timestamp. If you want to return a list of events sorted by username and then with the most recent events first. To create this index, use the following command:

```
db.events.ensureIndex( { "username" : 1, "timestamp" : -1 } )
```

Multikey

If you index a field that contains an array, you will create a multikey index, which adds entries to the index for *every* item in the array. 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 will add items to the index for all of the sub-documents in the array. As a result you will be able to run the following query, using only the index to locate the document:

```
db.feedback.find( { "comments.text": "Please expand the olive selection." } )
```

Note: To build or rebuild indexes for a *replica set* see *Building Indexes on Replica Sets* (page 241).

Warning: MongoDB will refuse to insert documents into a compound index where more than one field is an array (i.e. `{a: [1, 2], b: [1, 2]}`); however, MongoDB permits documents in collections with compound indexes where only one field per compound index is an array (i.e. `{a: [1, 2], b: 1}` and `{a: 1, b: [1, 2]}`.)

Unique Index

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 } )
```

If you use the unique constraint on a *compound index* (page 233) 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. MongoDB will only permit one document without a unique value in the collection because of this unique constraint. You can combine with the *sparse index* (page 234) to filter these null values from the unique index.

Sparse Index

Sparse indexes only contain entries for documents that have the indexed field. [3] By contrast, non-sparse indexes contain all documents in a collection, and store null values for documents that do not contain the indexed field. Create a sparse index on the `xmpp_id` field, of the `members` collection, using the following operation in the `mongo` shell:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

[3] All documents that have the indexed field *are* indexed in a sparse index, even if that field stores a null value in some documents.

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 indexes in other databases. Think of them as dense indexes with a specific filter.

You can combine the sparse index option with the *unique indexes* (page 234) option so that `mongod` will reject documents that have duplicate values for a field, but that ignore documents that do not have the key.

18.1.3 Index Creation Options

Most parameters [4] to the `ensureIndex()` operation affect the kind of index that MongoDB creates. Two options, *background construction* (page 235) and *duplicate dropping* (page 236), affect how MongoDB builds the indexes.

Background Construction

By default, creating an index is a blocking operation. Building an index on a large collection of data, the operation can take a long time to complete. To resolve this issue, the background option can allow you to continue to use your `mongod` instance during the index build. Create an index in the background of the `zipcode` field of the `people` collection using a command that resembles the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

You can combine the background option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true, sparse: true } )
```

Be aware of the following behaviors with background index construction:

- A `mongod` instance can 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.
- The indexing operation runs 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. Open another connection or `mongo` instance to continue using commands to the database.
- The background index operation use 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.

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.

[4] Other functionality accessible by way of parameters include *sparse* (page 234), *unique* (page 234), and *TTL* (page 236).

Remember, the amount of time required to build the index on a secondary node must be within the window of the *oplog*, so that the secondary can catch up with the primary.

See *Building Indexes on Replica Sets* (page 241) for more information on this process.

Indexes on secondary members in "recovering" mode are always built in the foreground to allow them to catch up as soon as possible.

Note: Administrative operations such as `repairDatabase` and `compact` will not run concurrently with a background index build.

Queries will not use these indexes until the index build is complete.

Duplicate Dropping

MongoDB cannot create a *unique index* (page 234) 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.

Warning: 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 234) 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.

18.1.4 Index Features

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 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.

Note: TTL indexes expire data by removing documents in a background task that runs once a minute. As a result, the TTL index provides no guarantees that expired documents will not exist in the collection. Consider that:

- Documents may remain in a collection *after* they expire and before the background process runs.
- The duration of the removal operations depend on the workload of your `mongod` instance.

In all other respects, TTL indexes are normal *secondary indexes* (page 232), and if appropriate, MongoDB can use these indexes to fulfill arbitrary queries.

See Also:

Expire Data from Collections by Setting TTL (page 296)

Geospatial Indexes

MongoDB provides "geospatial indexes" to support location-based and other similar queries in a two dimensional coordinate systems. For example, use geospatial indexes when you need to take a collection of documents that have coordinates, and return a number of options that are "near" a given coordinate pair.

To create a geospatial index, your *documents* must have a coordinate pair. For maximum compatibility, these coordinate pairs should be in the form of a two element array, such as `[ x , y ]`. Given the field of `loc`, that held a coordinate pair, in the collection `places`, you would create a geospatial index as follows:

```
db.places.ensureIndex( { loc : "2d" } )
```

MongoDB will reject documents that have values in the `loc` field beyond the minimum and maximum values.

Note: MongoDB permits only one geospatial index per collection. Although, MongoDB will allow clients to create multiple geospatial indexes, a single query can use only one index.

See the `$near`, and the database command `geoNear` for more information on accessing geospatial data.

Geohaystack Indexes

In addition to conventional *geospatial indexes* (page 237), MongoDB also provides a bucket-based geospatial index, called "geospatial haystack indexes." These indexes support high performance queries for locations within a small area, when the query must filter along another dimension.

Example

If you need to return all documents that have coordinates within 25 miles of a given point *and* have a type field value of "museum," a haystack index would be provide the best support for these queries.

Haystack indexes allow you to tune your bucket size to the distribution of your data, so that in general you search only very small regions of 2d space for a particular kind of document. These indexes are not suited for finding the closest documents to a particular location, when the closest documents are far away compared to bucket size.

18.1.5 Index Limitations

Be aware of the following current limitations of MongoDB's indexes:

- A collection may have no more than *64 indexes* (page 680).
- Index keys can be no larger than *1024 bytes* (page 680).

  This includes the field value or values, the field name or names, and the *namespace*.

- The name of an index, including the *namespace* must be shorter than *128 characters* (page 680).
- Indexes have storage requirements, and impacts insert/update speed to some degree.
- Create indexes to support queries and other operations, but do not maintain indexes that your MongoDB instance cannot or will not use.

18.2 Indexing Operations

18.2.1 Synopsis

This document provides operational guidelines and procedures for indexing data in MongoDB collections. For the fundamentals of MongoDB indexing, see the *Indexing Overview* (page 231) document. For strategies and practical approaches, see the *Indexing Strategies* (page 242) document.

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

18.2.2 Operations

Create an Index

To create an index, use `db.collection.ensureIndex()` or a similar method from your driver. For example the following creates [5] an index on the `phone-number` field of the `people` collection:

```
db.people.ensureIndex( { "phone-number": 1 } )
```

To create a *compound index* (page 233), use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

For 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 } )
```

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

Note: To build or rebuild indexes for a *replica set* see *Building Indexes on Replica Sets* (page 241).

Special Creation Options

Note: TTL collections use a special `expire` index option. See *Expire Data from Collections by Setting TTL* (page 296) for more information.

---

[5] As the name suggests, `ensureIndex()` only creates an index if an index of the same specification does not already exist.

Sparse Indexes

To create a *sparse index* (page 234) on a field, use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

The following example creates a sparse index on the `users` table that *only* indexes the `twitter_name` *if* a document has this field. This index will not include documents in this collection without the `twitter_name` field.

```
db.users.ensureIndex( { twitter_name: 1 }, { sparse: true } )
```

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 234) section for more information.

Unique Indexes

To create a *unique indexes* (page 234), 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 232) 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 233), 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.

Background

To create an index in the background you can specify *background construction* (page 235). Consider the following prototype invocation of `db.collection.ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on *background index construction* (page 235) for more information about these indexes and their implications.

Drop Duplicates

To force the creation of a *unique index* (page 234) 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 `db.collection.ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { dropDups: true } )
```

See the full documentation of *duplicate dropping* (page 236) 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.

List a Collection's Indexes

To list a collection's indexes, use the `db.collection.getIndexes()` method or a similar method for your driver.

For example, to view all indexes on the the `people` collection:

```
db.people.getIndexes()
```

Remove an Index

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 232) from a collection.

These shell helpers provide wrappers around the `dropIndexes` *database command*. Your *client library* (page 285) may have a different or additional interface for these operations.

Rebuilding

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method. This will drop all indexes, including the *_id index* (page 232), and then rebuild all indexes. 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 285) may have a different or additional interface for this operation.

Note: To build or rebuild indexes for a *replica set* see *Building Indexes on Replica Sets* (page 241).

18.2.3 Building Indexes on Replica Sets

Consideration

Background index creation operations (page 235) 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` (page 676) 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:

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*.

---

1. Stop the `mongod` process on one secondary. Restart the `mongod` process *without* the `--replSet` (page 586) option and running on a different port. [6] This instance is now in "standalone" mode.
2. Create the new index or rebuild the index on this `mongod` instance.
3. Restart the `mongod` instance with the `--replSet` (page 586) option. Allow replication to catch up on this member.
4. Repeat this operation on all of the remaining secondaries.
5. Run `rs.stepDown()` on the *primary* member of the set, and then repeat this procedure on the former primary.

[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.

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 36)" 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.

18.2.4 Measuring Index Use

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides the following tools:

- `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.

- `cursor.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 } )
  ```

- `indexCounters` (page 643)

  Use the `indexCounters` (page 643) data in the output of `serverStatus` for insight into database-wise index utilization.

18.2.5 Monitoring and Controlling Index Building

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.

You can only terminate a background index build. If you need to terminate an ongoing index build, You can use the `db.killOp()` method in the `mongo` shell.

18.3 Indexing Strategies

This document provides strategies for indexing in MongoDB. For fundamentals of MongoDB indexing, see *Indexing Overview* (page 231). For operational guidelines and procedures, see *Indexing Operations* (page 238).

18.3.1 Strategies

The best indexes for your application are based on a number of factors, 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:

- The application's queries.
- The relative frequency of each query in the application.
- The current indexes created for your collections.
- Which indexes the most common queries use.

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 and to see which configurations perform best.

MongoDB can only use *one* index to support any given operation. However, each clause of an `$or` query can use its own index.

18.3.2 Create Indexes to Support Your Queries

If you only ever query on a single key in a given collection, then you need 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 } )
```

However, if you sometimes query on only one key but and at other times query on that key combined with a second key, then creating a *compound index* (page 233) is more efficient. 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`. (To query on multiple keys and sort the results, see *Use Indexes to Sort Query Results* (page 244).)

With the exception of queries that use the `$or` operator, a query cannot use multiple indexes. A query must use only one index.

18.3.3 Use Compound Indexes to Support Several Different Queries

A single *compound index* (page 233) on multiple fields can support all the queries that search a "prefix" subset of those fields.

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 244).

18.3.4 Create Indexes that Support Covered Queries

A covered index query is a query in which all the queried fields are part of an index. They are "covered queries" because an index "covers" the query. MongoDB can fulfill the query by using *only* the index. MongoDB need not scan documents from the database.

Querying *only* the index is much faster than querying documents. Indexes keys are typically smaller than the documents they catalog, and indexes are typically stored in RAM or located sequentially on disk.

Mongod automatically uses a covered query when possible. To ensure use of a covered query, create an index that includes all the fields listed in the query result. This means that the *projection* document given to a query (to specify which fields MongoDB returns from the result set) must explicitly exclude the `_id` field from the result set, unless the index includes `_id`.

MongoDB cannot use a covered query if any of the indexed fields in any of the documents in the collection include an array. If an indexed field is an array, the index becomes a *multi-key index* (page 233) index and cannot support a covered query.

To test whether MongoDB used a covered query, use `explain()`. If the output displays `true` for the `indexOnly` field, MongoDB used a covered query. For more information see *Measuring Index Use* (page 242).

18.3.5 Use Indexes to Sort Query Results

For the fastest performance when sorting query results by a given field, create a sorted index on that field.

To sort query results on multiple fields, create a *compound index* (page 233). MongoDB sorts results based on the field order in the index. For queries that include a sort that uses a compound index, ensure that all fields before the first sorted field are equality matches.

Example

If you create the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following query and sort operations can use the index:

```
db.collection.find().sort( { a:1 } )
db.collection.find().sort( { a:1, b:1 } )

db.collection.find( { a:4 } ).sort( { a:1, b:1 } )
db.collection.find( { b:5 } ).sort( { a:1, b:1 } )

db.collection.find( { a:5 } ).sort( { b:1, c:1 } )
```

```
db.collection.find( { a:5, c:4, b:3 } ).sort( { d:1 } )

db.collection.find( { a: { $gt:4 } } ).sort( { a:1, b:1 } )
db.collection.find( { a: { $gt:5 } } ).sort( { a:1, b:1 } )

db.collection.find( { a:5, b:3, d:{ $gt:4 } } ).sort( { c:1 } )
db.collection.find( { a:5, b:3, c:{ $lt:2 }, d:{ $gt:4 } } ).sort( { c:1 } )
```

However, the following queries cannot sort the results using the index:

```
db.collection.find().sort( { b:1 } )
db.collection.find( { b:5 } ).sort( { b:1 } )
```

Note: When the `sort()` method performs an in-memory sort operation without the use of an index, the operation is significantly slower than for operations that use an index, and the operation aborts when it consumes 32 megabytes of memory.

18.3.6 Ensure Indexes Fit RAM

For 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 RAM at the same time.

There are some limited cases where indexes do not need to fit in RAM. See *Indexes that Hold Only Recent Values in RAM* (page 245).

See Also:

For additional *collection statistics* (page 653), use `collStats` or `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 grows 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.

18.3.7 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 233) 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 *Measuring Index Use* (page 242).

18.3.8 Consider Performance when Creating Indexes for Write-heavy Applications

If your application is write-heavy, then be careful when creating new indexes, since each additional index with impose a write-performance penalty. In general, don't be careless about adding indexes. Add indexes complement your queries. Always have a good reason for adding a new index, and make sure you've benchmarked alternative strategies.

Consider Insert Throughput

MongoDB must update *all* indexes associated with a collection after every insert, update, or delete operation. For update operations, if the updated document does not move to a new location, then only the modified, MongoDB only needs to update the updated fields in the index. Therefore, every index on a collection adds some amount of overhead to these write operations. In almost every case, the performance gains that indexes realize for read operations are worth the insertion penalty. However, in some cases:

- An index to support an infrequent query might incur more insert-related costs than saved read-time.
- If you have many indexes on a collection with a high insert throughput and a number of related indexes, you may find better overall performance with a smaller number of indexes, even if some queries are less optimally supported by an index.
- If your indexes and queries are not sufficiently *selective* (page 245), the speed improvements for query operations might not offset the costs of maintaining an index. For more information see *Create Queries that Ensure Selectivity* (page 245).

Part VIII

Aggregation

Aggregation provides a natural method for aggregating data inside of MongoDB.

For a description of MongoDB aggregation, see *Aggregation Framework* (page 253). For examples of aggregation, see *Aggregation Framework Examples* (page 259). For descriptions of aggregation operators, see *Aggregation Framework Reference* (page 269).

The following is the outline of the aggregation documentation:

CHAPTER

NINETEEN

AGGREGATION FRAMEWORK

New in version 2.1.

19.1 Overview

The MongoDB aggregation framework provides a means to calculate aggregated values without having to use *map-reduce*. While map-reduce is powerful, it is often more difficult than necessary for many simple aggregation tasks, such as totaling or averaging field values.

If you're familiar with *SQL*, the aggregation framework provides similar functionality to `GROUP BY` and related SQL operators as well as simple forms of "self joins." Additionally, the aggregation framework provides projection capabilities to reshape the returned data. Using the projections in the aggregation framework, you can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results.

See Also:

A presentation from MongoSV 2011: MongoDB's New Aggregation Framework.

Additionally, consider *Aggregation Framework Examples* (page 259) and *Aggregation Framework Reference* (page 269) for additional documentation.

19.2 Framework Components

This section provides an introduction to the two concepts that underpin the aggregation framework: *pipelines* and *expressions*.

19.2.1 Pipelines

Conceptually, documents from a collection pass 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. `|`) used to string text filters together.

In a shell environment the pipe redirects a stream of characters from the output of one process to the input of the next. The MongoDB aggregation pipeline streams MongoDB documents from one *pipeline operator* (page 270) to the next to process the documents.

All pipeline operators process a stream of documents and the pipeline behaves as if the operation scans a *collection* and passes all matching documents into the "top" of the pipeline. Each operator in the pipeline transforms each document as it passes through the pipeline.

Note: Pipeline operators need not produce one output document for every input document: operators may also generate new documents or filter out documents.

Warning: The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

See Also:

The "*Aggregation Framework Reference* (page 269)" includes documentation of the following pipeline operators:

- `$project` (page 477)
- `$match` (page 475)
- `$limit` (page 475)
- `$skip` (page 479)
- `$unwind` (page 481)
- `$group` (page 473)
- `$sort` (page 479)

19.2.2 Expressions

Expressions (page 276) produce output documents based on calculations performed on input documents. The aggregation framework defines expressions using a document format using prefixes.

Expressions are stateless and are only evaluated when seen by the aggregation process. All aggregation expressions can only operate on the current document in the pipeline, and cannot integrate data from other documents.

The *accumulator* expressions used in the `$group` (page 473) operator maintain that state (e.g. totals, maximums, minimums, and related data) as documents progress through the *pipeline*.

See Also:

Aggregation expressions (page 276) for additional examples of the expressions provided by the aggregation framework.

19.3 Use

19.3.1 Invocation

Invoke an *aggregation* operation with the `aggregate()` wrapper in the `mongo` shell or the `aggregate` *database command*. Always call `aggregate()` on a collection object that determines the input documents of the aggregation *pipeline*. The arguments to the `aggregate()` method specify a sequence of *pipeline operators* (page 270), where each operator may have a number of operands.

First, consider a *collection* of documents named `articles` using the following format:

```
{
 title : "this is my title" ,
 author : "bob" ,
 posted : new Date () ,
 pageViews : 5 ,
 tags : [ "fun" , "good" , "fun" ] ,
 comments : [
             { author :"joe" , text : "this is cool" } ,
```

```
            { author :"sam" , text : "this is bad" }
 ],
 other : { foo : 5 }
}
```

The following example aggregation operation pivots data to create a set of author names grouped by tags applied to an article. Call the aggregation framework by issuing the following command:

```
db.articles.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : { tags : "$tags" },
     authors : { $addToSet : "$author" }
  } }
);
```

The aggregation pipeline begins with the *collection* `articles` and selects the `author` and `tags` fields using the `$project` (page 477) aggregation operator. The `$unwind` (page 481) operator produces one output document per tag. Finally, the `$group` (page 473) operator pivots these fields.

19.3.2 Result

The aggregation operation in the previous section returns a *document* with two fields:

- `result` which holds an array of documents returned by the *pipeline*
- `ok` which holds the value `1`, indicating success, or another value if there was an error

As a document, the result is subject to the *BSON Document size* (page 679) limit, which is currently 16 megabytes.

19.4 Optimizing Performance

Because you will always call `aggregate` on a *collection* object, which logically inserts the *entire* collection into the aggregation pipeline, you may want to optimize the operation by avoiding scanning the entire collection whenever possible.

19.4.1 Pipeline Operators and Indexes

Depending on the order in which they appear in the pipeline, aggregation operators can take advantage of indexes.

The following pipeline operators take advantage of an index when they occur at the beginning of the pipeline:

- `$match` (page 475)
- `$sort` (page 479)
- `$limit` (page 475)
- `$skip` (page 479).

The above operators can also use an index when placed **before** the following aggregation operators:

- `$project` (page 477)

- $unwind (page 481)
- $group (page 473).

19.4.2 Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the $match (page 475) operator to restrict which items go in to the top of the pipeline, as in a query. When placed early in a pipeline, these $match (page 475) operations use suitable indexes to scan only the matching documents in a collection.

Placing a $match (page 475) pipeline stage followed by a $sort (page 479) stage at the start of the pipeline is logically equivalent to a single query with a sort, and can use an index.

In future versions there may be an optimization phase in the pipeline that reorders the operations to increase performance without affecting the result. However, at this time place $match (page 475) operators at the beginning of the pipeline when possible.

19.4.3 Memory for Cumulative Operators

Certain pipeline operators require access to the entire input set before they can produce any output. For example, $sort (page 479) must receive all of the input from the preceding *pipeline* operator before it can produce its first output document. The current implementation of $sort (page 479) does not go to disk in these cases: in order to sort the contents of the pipeline, the entire input must fit in memory.

$group (page 473) has similar characteristics: Before any $group (page 473) passes its output along the pipeline, it must receive the entirety of its input. For the $group (page 473) operator, this frequently does not require as much memory as $sort (page 479), because it only needs to retain one record for each unique key in the grouping specification.

The current implementation of the aggregation framework logs a warning if a cumulative operator consumes 5% or more of the physical memory on the host. Cumulative operators produce an error if they consume 10% or more of the physical memory on the host.

19.5 Sharded Operation

Note: Changed in version 2.1. Some aggregation operations using aggregate will cause mongos (page 676) instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architectural decisions if you use the *aggregation framework* extensively in a sharded environment.

The aggregation framework is compatible with sharded collections.

When operating on a sharded collection, the aggregation pipeline is split into two parts. The aggregation framework pushes all of the operators up to the first $group (page 473) or $sort (page 479) operation to each shard. [1] Then, a second pipeline on the mongos (page 676) runs. This pipeline consists of the first $group (page 473) or $sort (page 479) and any remaining pipeline operators, and runs on the results received from the shards.

The $group (page 473) operator brings in any "sub-totals" from the shards and combines them: in some cases these may be structures. For example, the $avg (page 474) expression maintains a total and count for each shard; mongos (page 676) combines these values and then divides.

[1] If an early $match (page 475) can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.

19.6 Limitations

Aggregation operations with the `aggregate` command have the following limitations:

- The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, `CodeWScope`.
- Output from the *pipeline* can only contain 16 megabytes. If your result set exceeds this limit, the `aggregate` command produces an error.
- If any single aggregation operation consumes more than 10 percent of system RAM the operation will produce an error.

CHAPTER

TWENTY

AGGREGATION FRAMEWORK EXAMPLES

MongoDB provides flexible data aggregation functionality with the `aggregate` command. For additional information about aggregation consider the following resources:

- *Aggregation Framework* (page 253)
- *Aggregation Framework Reference* (page 269)
- SQL+to+Aggregation+Framework+Mapping+Chart

This document provides a number of practical examples that display the capabilities of the aggregation framework. All examples use a publicly available data set of all zipcodes and populations in the United States.

20.1 Requirements

`mongod` and `mongo`, version 2.2 or later.

20.2 Aggregations using the Zip Code Data Set

To run you will need the zipcode data set. These data are available at: media.mongodb.org/zips.json. Use `mongoimport` to load this data set into your `mongod` instance.

20.2.1 Data Model

Each document in this collection has the following form:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

In these documents:

- The _id field holds the zipcode 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 285) for a more idiomatic interface for data aggregation operations.

20.2.2 States with Populations Over 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 on the zipcodes collection. aggregate() a number of *pipeline* (page 253) operators that define the aggregation process.

In the above example, the pipeline passes all documents in the zipcodes collection through the following steps:

- the $group (page 473) 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 (page 481) operation to calculate the total value of all pop fields in the source documents.

  After the $group (page 473) operation the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```

- the $match (page 475) 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 (page 475) operation does not alter the documents, which have the same format as the documents output by $group (page 473).

The equivalent *SQL* for this operation is:

```
SELECT state, SUM(pop) AS pop
       FROM zips
       GROUP BY state
       HAVING pop > (10*1000*1000)
```

20.2.3 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 on the `zipcodes` collection. `aggregate()` a number of *pipeline* (page 253) operators that define the aggregation process.

In the above example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` (page 473) 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 `$group` (page 473) operator collects documents by the `state` field and use the `$avg` (page 474) expression to compute a value for the `avgCityPop` field.

The final output of this aggregation operation is:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
},
```

20.2.4 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" } } },

                       // the following $project is optional, and
                       // modifies the output format.

                       { $project:
                         { _id: 0,
                           state: "$_id",
                           biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
                           smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } } )
```

Aggregations operations using the `aggregate()` helper, process all documents on the `zipcodes` collection. `aggregate()` a number of *pipeline* (page 253) operators that define the aggregation process.

All documents from the zipcodes collection pass into the pipeline, which consists of the following steps:

- the $group (page 473) 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 (page 481) 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 (page 479) 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 (page 473) 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 (page 473) operator specifies four fields: Using the $last (page 475) expression, the $group (page 473) operator creates the biggestcity and biggestpop fields that store the city with the largest population and that population. Using the $first (page 473) expression, the $group (page 473) 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 (page 477), which renames the _id field to state and moves the biggestCity, biggestPop, smallestCity, and smallestPop into biggestCity and smallestCity sub-documents.

The final output of this aggregation operation is:

```
{
  "state" : "RI",
  "biggestCity" : {
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
    "pop" : 45
  }
}
```

20.3 Aggregation with User Preference Data

20.3.1 Data Model

Consider a hypothetical sports club with a database that contains a `user` collection that tracks 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"]
}
```

20.3.2 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 passes through the pipeline, which consists of the following operations:

- The `$project` (page 477) operator:
  - creates a new field called `name`.
  - converts the value of the `_id` to upper case, with the `$toUpper` (page 481) operator. Then the `$project` (page 477) creates a new filed, named `name` to hold this value.
  - suppresses the `id` field. `$project` (page 477) will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` (page 479) operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

20.3.3 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 `$project` (page 477) operator:
  - Creates two new fields: `month_joined` and `name`.
  - Suppresses the `id` from the results. The `aggregate()` method includes the `_id`, unless explicitly suppressed.
- The `$month` (page 477) operator converts the values of the `joined` field to integer representations of the month. Then the `$project` (page 477) operator assigns those values to the `month_joined` field.
- The `$sort` (page 479) 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"
}
```

20.3.4 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 such information 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 `$project` (page 477) operator creates a new field called `month_joined`.
- The `$month` (page 477) operator converts the values of the `joined` field to integer representations of the month. Then the `$project` (page 477) operator assigns the values to the `month_joined` field.
- The `$group` (page 473) 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, `$group` (page 473) creates a new "per-month" document with two fields:
  - `_id`, which contains a nested document with the `month_joined` field and its value.
  - `number`, which is a generated field. The `$sum` (page 481) operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` (page 479) operator sorts the documents created by `$group` (page 473) 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
}
```

20.3.5 Return the Five Most Common "Likes"

The following aggregation collects top five most "liked" activities in the data set. In this data set, you might use an analysis of this to 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` (page 481) 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` (page 481) 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 `$group` (page 473) operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` (page 473) creates a new document with two fields:
  - `_id`, which contains the `likes` value.
  - `number`, which is a generated field. The `$sum` (page 481) operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` (page 479) operator sorts these documents by the `number` field in reverse order.
- The `$limit` (page 475) 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
}
```

CHAPTER

TWENTYONE

AGGREGATION FRAMEWORK REFERENCE

New in version 2.1.0. The aggregation framework provides the ability to project, process, and/or control the output of the query, without using *map-reduce*. Aggregation uses a syntax that resembles the same syntax and form as "regular" MongoDB database queries.

These aggregation operations are all accessible by way of the `aggregate()` method. While all examples in this document use this method, `aggregate()` is merely a wrapper around the *database command* `aggregate`. The following prototype aggregation operations are equivalent:

```
db.people.aggregate( <pipeline> )
db.people.aggregate( [<pipeline>] )
db.runCommand( { aggregate: "people", pipeline: [<pipeline>] } )
```

These operations perform aggregation routines on the collection named `people`. `<pipeline>` is a placeholder for the aggregation *pipeline* definition. `aggregate()` accepts the stages of the pipeline (i.e. `<pipeline>`) as an array, or as arguments to the method.

This documentation provides an overview of all aggregation operators available for use in the aggregation pipeline as well as details regarding their use and behavior.

See Also:

Aggregation Framework (page 253) overview, the *Aggregation Framework Documentation Index* (page 251), and the *Aggregation Framework Examples* (page 259) for more information on the aggregation functionality.

Aggregation Operators:

21.1 Pipeline

Warning: The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

Pipeline operators appear in an array. Conceptually, documents pass through these operators in a sequence. All examples in this section assume that the aggregation pipeline begins with a collection named `article` that contains documents that resemble the following:

```
{
  title : "this is my title" ,
  author : "bob" ,
  posted : new Date() ,
  pageViews : 5 ,
  tags : [ "fun" , "good" , "fun" ] ,
  comments : [
      { author :"joe" , text : "this is cool" } ,
      { author :"sam" , text : "this is bad" }
  ],
  other : { foo : 5 }
}
```

The current pipeline operators are:

$project

Reshapes a document stream by renaming, adding, or removing fields. Also use `$project` (page 477) to create computed values or sub-objects. Use `$project` (page 477) to:

- Include fields from the original document.
- Insert computed fields.
- Rename fields.
- Create and populate fields that hold sub-documents.

Use `$project` (page 477) to quickly select the fields that you want to include or exclude from the response. Consider the following aggregation framework operation.

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
 );
```

This operation includes the `title` field and the `author` field in the document that returns from the aggregation *pipeline*.

Note: The `_id` field is always included by default. You may explicitly exclude `_id` as follows:

```
db.article.aggregate(
    { $project : {
        _id : 0 ,
        title : 1 ,
        author : 1
    }}
);
```

Here, the projection excludes the _id field but includes the title and author fields.

Projections can also add computed fields to the document stream passing through the pipeline. A computed field can use any of the *expression operators* (page 276). Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1,
        doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

Here, the field doctoredPageViews represents the value of the pageViews field after adding 10 to the original field using the $add (page 471).

Note: You must enclose the expression that defines the computed field in braces, so that the expression is a valid object.

You may also use $project (page 477) to rename fields. Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        page_views : "$pageViews" ,
        bar : "$other.foo"
    }}
);
```

This operation renames the pageViews field to page_views, and renames the foo field in the other sub-document as the top-level field bar. The field references used for renaming fields are direct expressions and do not use an operator or surrounding braces. All aggregation field references can use dotted paths to refer to fields in nested documents.

Finally, you can use the $project (page 477) to create and populate new sub-documents. Consider the following example that creates a new object-valued field named stats that holds a number of values:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {
            pv : "$pageViews",
            foo : "$other.foo",
            dpv : { $add:["$pageViews", 10] }
        }
    }}
);
```

This projection includes the title field and places $project (page 477) into "inclusive" mode. Then, it creates the stats documents with the following fields:

- pv which includes and renames the pageViews from the top level of the original documents.
- foo which includes the value of other.foo from the original documents.
- dpv which is a computed field that adds 10 to the value of the pageViews field in the original document using the $add (page 471) aggregation expression.

$match

Provides a query-like interface to filter documents out of the aggregation *pipeline*. The $match (page 475)

drops documents that do not match the condition from the aggregation pipeline, and it passes documents that match along the pipeline unaltered.

The syntax passed to the $match (page 475) is identical to the *query* syntax. Consider the following prototype form:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

The following example performs a simple field equality test:

```
db.article.aggregate(
    { $match : { author : "dave" } }
);
```

This operation only returns documents where the author field holds the value dave. Consider the following example, which performs a range test:

```
db.article.aggregate(
    { $match : { score  : { $gt : 50, $lte : 90 } } }
);
```

Here, all documents return when the score field holds a value that is greater than 50 and less than or equal to 90.

Note: Place the $match (page 475) as early in the aggregation *pipeline* as possible. Because $match (page 475) limits the total number of documents in the aggregation pipeline, earlier $match (page 475) operations minimize the amount of later processing. If you place a $match (page 475) at the very beginning of a pipeline, the query can take advantage of *indexes* like any other db.collection.find() or db.collection.findOne().

Warning: You cannot use $where or *geospatial operations* in $match (page 475) queries as part of the aggregation pipeline.

$limit

Restricts the number of *documents* that pass through the $limit (page 475) in the *pipeline*.

$limit (page 475) takes a single numeric (positive whole number) value as a parameter. Once the specified number of documents pass through the pipeline operator, no more will. Consider the following example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit (page 475) has no effect on the content of the documents it passes.

$skip

Skips over the specified number of *documents* that pass through the $skip (page 479) in the *pipeline* before passing all of the remaining input.

$skip (page 479) takes a single numeric (positive whole number) value as a parameter. Once the operation has skipped the specified number of documents, it passes all the remaining documents along the *pipeline* without alteration. Consider the following example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This operation skips the first 5 documents passed to it by the pipeline. $skip (page 479) has no effect on the content of the documents it passes along the pipeline.

$unwind

Peels off the elements of an array individually, and returns a stream of documents. $unwind (page 481) returns one document for every member of the unwound array within every source document. Take the following aggregation command:

```
db.article.aggregate(
    { $project : {
        author : 1 ,
        title : 1 ,
        tags : 1
    }},
    { $unwind : "$tags" }
);
```

Note: The dollar sign (i.e. $) must proceed the field specification handed to the $unwind (page 481) operator.

In the above aggregation $project (page 477) selects (inclusively) the author, title, and tags fields, as well as the _id field implicitly. Then the pipeline passes the results of the projection to the $unwind (page 481) operator, which will unwind the tags field. This operation may return a sequence of documents that resemble the following for a collection that contains one document holding a tags field with an array of 3 items.

```
{
     "result" : [
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "good"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             }
     ],
     "OK" : 1
}
```

A single document becomes 3 documents: each document is identical except for the value of the tags field. Each value of tags is one of the values in the original "tags" array.

Note: $unwind (page 481) has the following behaviors:

- $unwind (page 481) is most useful in combination with $group (page 473).

- You may undo the effects of unwind operation with the $group (page 473) pipeline operator.
- If you specify a target field for $unwind (page 481) that does not exist in an input document, the pipeline ignores the input document, and will generate no result documents.
- If you specify a target field for $unwind (page 481) that is not an array, aggregate() generates an error.
- If you specify a target field for $unwind (page 481) that holds an empty array ([]) in an input document, the pipeline ignores the input document, and will generates no result documents.

$group

Groups documents together for the purpose of calculating aggregate values based on a collection of documents. Practically, group often supports tasks such as average page views for each page in a website on a daily basis.

The output of $group (page 473) depends on how you define groups. Begin by specifying an identifier (i.e. a _id field) for the group you're creating with this pipeline. You can specify a single field from the documents in the pipeline, a previously computed value, or an aggregate key made up from several incoming fields. Aggregate keys may resemble the following document:

```
{ _id : { author: '$author', pageViews: '$pageViews', posted: '$posted' } }
```

With the exception of the _id field, $group (page 473) cannot output nested documents.

Every group expression must specify an _id field. You may specify the _id field as a dotted field path reference, a document with multiple fields enclosed in braces (i.e. { and }), or a constant value.

Note: Use $project (page 477) as needed to rename the grouped field after an $group (page 473) operation, if necessary.

Consider the following example:

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

This groups by the author field and computes two fields, the first docsPerAuthor is a counter field that adds one for each document with a given author field using the $sum (page 481) function. The viewsPerAuthor field is the sum of all of the pageViews fields in the documents for each group.

Each field defined for the $group (page 473) must use one of the group aggregation function listed below to generate its composite value:

$addToSet

Returns an array of all the values found in the selected field among the documents in that group. *Every unique value only appears once* in the result set. There is no ordering guarantee for the output documents.

$first

Returns the first value it encounters for its group .

Note: Only use $first (page 473) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$last
Returns the last value it encounters for its group.

Note: Only use $last (page 475) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$max
Returns the highest value among all values of the field in all documents selected by this group.

$min
Returns the lowest value among all values of the field in all documents selected by this group.

$avg
Returns the average of all the values of the field in all documents selected by this group.

$push
Returns an array of all the values found in the selected field among the documents in that group. *A value may appear more than once* in the result set if more than one field in the grouped documents has that value.

$sum
Returns the sum of all the values for a specified field in the grouped documents, as in the second use above.

Alternately, if you specify a value as an argument, $sum (page 481) will increment this field by the specified value for every document in the grouping. Typically, as in the first use above, specify a value of 1 " in order to count members of the group.

Warning: The aggregation system currently stores $group (page 473) operations in memory, which may cause problems when processing a larger number of groups.

$sort
The $sort (page 479) *pipeline* operator sorts all input documents and returns them to the pipeline in sorted order. Consider the following prototype form:

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

This sorts the documents in the collection named <collection-name>, according to the key and specification in the { <sort-key> } document.

Specify the sort in a document with a field or fields that you want to sort by and a value of 1 or -1 to specify an ascending or descending sort respectively, as in the following example:

```
db.users.aggregate(
    { $sort : { age : -1, posts: 1 } }
);
```

This operation sorts the documents in the users collection, in descending order according by the age field and then in ascending order according to the value in the posts field.

Note: The $sort (page 479) cannot begin sorting documents until previous operators in the pipeline have returned all output.

- $skip (page 479)

$sort (page 479) operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators:

- $project (page 477)
- $unwind (page 481)
- $group (page 473).

Warning: Unless the $sort (page 479) operator can use an index, in the current release, the sort must fit within memory. This may cause problems when sorting large numbers of documents.

21.2 Expressions

These operators calculate values within the *aggregation framework*.

21.2.1 Boolean Operators

The three boolean operators accept Booleans as arguments and return Booleans as results.

Note: These operators convert non-booleans to Boolean values according to the BSON standards. Here, null, undefined, and 0 values become false, while non-zero numeric values, and all other types, such as strings, dates, objects become true.

$and

Takes an array one or more values and returns true if *all* of the values in the array are true. Otherwise $and returns false.

Note: $and uses short-circuit logic: the operation stops evaluation after encountering the first false expression.

$or

Takes an array of one or more values and returns true if *any* of the values in the array are true. Otherwise $or returns false.

Note: $or uses short-circuit logic: the operation stops evaluation after encountering the first true expression.

$not

Returns the boolean opposite value passed to it. When passed a true value, $not returns false; when passed a false value, $not returns true.

21.2.2 Comparison Operators

These operators perform comparisons between two values and return a Boolean, in most cases, reflecting the result of that comparison.

All comparison operators take an array with a pair of values. You may compare numbers, strings, and dates. Except for $cmp (page 471), all comparison operators return a Boolean value. $cmp (page 471) returns an integer.

$cmp

Takes two values in an array and returns an integer. The returned value is:

- A negative number if the first value is less than the second.

- A positive number if the first value is greater than the second.
- `0` if the two values are equal.

`$eq`
Takes two values in an array and returns a boolean. The returned value is:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

`$gt`
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than* the second value.
- `false` when the first value is *less than or equal to* the second value.

`$gte`
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than or equal* to the second value.
- `false` when the first value is *less than* the second value.

`$lt`
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than* the second value.
- `false` when the first value is *greater than or equal to* the second value.

`$lte`
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than or equal to* the second value.
- `false` when the first value is *greater than* the second value.

`$ne`
Takes two values in an array returns an integer. The returned value is:

- `true` when the values are **not equivalent**.
- `false` when the values are **equivalent**.

21.2.3 Arithmetic Operators

These operators only support numbers.

`$add`
Takes an array of one or more numbers and adds them together, returning the sum.

`$divide`
Takes an array that contains a pair of numbers and returns the value of the first number divided by the second number.

`$mod`
Takes an array that contains a pair of numbers and returns the *remainder* of the first number divided by the second number.

See Also:

`$mod`

$multiply
Takes an array of one or more numbers and multiples them, returning the resulting product.

$subtract
Takes an array that contains a pair of numbers and subtracts the second from the first, returning their difference.

21.2.4 String Operators

These operators manipulate strings within projection expressions.

$strcasecmp
Takes in two strings. Returns a number. $strcasecmp (page 480) is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. $strcasecmp (page 480) returns 0 if the strings are identical.

Note: $strcasecmp (page 480) may not make sense when applied to glyphs outside the Roman alphabet.

$strcasecmp (page 480) internally capitalizes strings before comparing them to provide a case-*insensitive* comparison. Use $cmp (page 471) for a case sensitive comparison.

$substr
$substr (page 480) takes a string and two numbers. The first number represents the number of bytes in the string to skip, and the second number specifies the number of bytes to return from the string.

Note: $substr (page 480) is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$toLower
Takes a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase.

Note: $toLower (page 481) may not make sense when applied to glyphs outside the Roman alphabet.

$toUpper
Takes a single string and converts that string to uppercase, returning the result. All lowercase letters become uppercase.

Note: $toUpper (page 481) may not make sense when applied to glyphs outside the Roman alphabet.

21.2.5 Date Operators

All date operators take a "Date" typed value as a single argument and return a number.

$dayOfYear
Takes a date and returns the day of the year as a number between 1 and 366.

$dayOfMonth
Takes a date and returns the day of the month as a number between 1 and 31.

$dayOfWeek
Takes a date and returns the day of the week as a number between 1 (Sunday) and 7 (Saturday.)

$year
Takes a date and returns the full year.

$month
Takes a date and returns the month as a number between 1 and 12.

$week
Takes a date and returns the week of the year as a number between 0 and 53.

Weeks begin on Sundays, and week 1 begins with the first Sunday of the year. Days preceding the first Sunday of the year are in week 0. This behavior is the same as the "%U" operator to the `strftime` standard library function.

$hour
Takes a date and returns the hour between 0 and 23.

$minute
Takes a date and returns the minute between 0 and 59.

$second
Takes a date and returns the second between 0 and 59, but can be 60 to account for leap seconds.

21.2.6 Conditional Expressions

$cond
Use the `$cond` (page 472) operator with the following syntax:

```
{ $cond: [ <boolean-expression>, <true-case>, <false-case> ] }
```

Takes an array with three expressions, where the first expression evaluates to a Boolean value. If the first expression evaluates to true, `$cond` (page 472) returns the value of the second expression. If the first expression evaluates to false, `$cond` (page 472) evaluates and returns the third expression.

$ifNull
Use the `$ifNull` (page 474) operator with the following syntax:

```
{ $ifNull: [ <expression>, <replacement-if-null> ] }
```

Takes an array with two expressions. `$ifNull` (page 474) returns the first expression if it evaluates to a non-null value. Otherwise, `$ifNull` (page 474) returns the second expression's value.

Part IX

Application Development

MongoDB provides language-specific client libraries called *drivers* that let you develop applications to interact with your databases.

This page lists the documents, tutorials, and reference pages that describe application development. For API-level documentation, see *Drivers* (page 285).

For an overview of topics with which every MongoDB application developer will want familiarity, see the *aggregation* (page 251) and *indexes* (page 229) documents. For an introduction to basic MongoDB use, see the *administration tutorials* (page 193).

See Also:

Developer Zone wiki pages and the *FAQ: MongoDB for Application Developers* (page 415) document. Developers also should be familiar with *Using the MongoDB Shell* (page 301) and the MongoDB `query and update operators`.

CHAPTER

TWENTYTWO

APPLICATION DEVELOPMENT

The following documents outline basic application development topics:

22.1 Drivers

Applications communicate with MongoDB by way of a client library or driver that handles all interaction with the database in language appropriate and sensible manner. See the following pages for more information about the MongoDB wiki drivers page:

- JavaScript (wiki, docs)
- Python (wiki, docs)
- Ruby (wiki, docs)
- PHP (wiki, docs)
- Perl (wiki, docs)
- Java (wiki, docs)
- Scala (wiki, docs)
- C# (wiki, docs)
- C (wiki, docs)
- C++ (wiki, docs)
- Haskell (wiki, docs)
- Erlang (wiki, docs)

22.2 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 286) 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 287) 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 queries to return the referenced documents. Many *drivers* (page 285) have helper methods that form the query for the DBRef automatically. The drivers [1] 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.

22.2.1 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 286). 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 287).

[1] Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

22.2.2 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.

Thus a DBRef document would resemble the following:

```
{ $ref : <value>, $id : <value>, $db : <value> }
```

Note: The order of fields in the DBRef matter, 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 Class and supplies the FetchDBRef Method for accessing these objects.

Java The DBRef 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 CPAN module.

PHP The PHP driver does support DBRefs, including the optional `$db` reference, through The MongoDBRef class.

Python The Python driver provides the DBRef class, and the dereference method for interacting with DBRefs.

Ruby The Ruby Driver supports DBRefs using the DBRef class and the deference method.

Use

In most cases you should use the *manual reference* (page 286) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

22.3 ObjectId

ObjectId is a 12-byte *BSON* type, constructed using:

- a 4-byte timestamp,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter.

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; i.e., the `mongod` adds the `_id` field and generates a unique ObjectId to assign as its value.

Using ObjectIds for the `_id` field, provides the following additional benefits:

- you can access the timestamp of the ObjectId's creation, using the `getTimestamp()` (page 527) method.
- Sorting on an `_id` field that stores ObjectId values, is equivalent to sorting by creation time.

22.3.1 ObjectId()

The `mongo` shell provides the `ObjectId()` wrapper class to generate can generate a new ObjectId, and to provide the following helper attribute and methods:

- `str`

  The hexadecimal string value of the `ObjectId()` object.

- `getTimestamp()` (page 527)

  Returns the timestamp portion of the `ObjectId()` object as a Date.

- `toString()` (page 527)

  Returns the string representation of the `ObjectId()` object. The returned string literal has the format "`ObjectId(...)`". Changed in version 2.2: In previous versions `ObjectId.toString()` (page 527) returns the value of the ObjectId as a hexadecimal string.

- `valueOf()` (page 528)

  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 `ObjectId.valueOf()` (page 528) returns the `ObjectId()` object.

Consider the example uses of the `ObjectId()` class in the `mongo` shell:

- 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()` (page 527) method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").getTimestamp()
  ```

  This operation will return the following Date object:

  ```
  ISODate("2012-10-17T20:46:22Z")
  ```

- 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 string representation of an `ObjectId()` object, use the `toString()` (page 527) method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").toString()
  ```

  This operation will return the following output:

  ```
  ObjectId("507f191e810c19729de860ea")
  ```

- To return the value of an `ObjectId()` object as a hexadecimal string, use the `valueOf()` (page 528) method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").valueOf()
  ```

  This operation will return the following output:

  ```
  507f191e810c19729de860ea
  ```

See Also:

- *Application Development with Replica Sets* (page 54)
- *Indexing Strategies* (page 242)
- *Aggregation Framework* (page 253)

CHAPTER

TWENTYTHREE

PATTERNS

The following documents provide patterns for developing application features:

23.1 Perform Two Phase Commits

23.1.1 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 294) like functionality.

23.1.2 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 294).

23.1.3 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, "
```

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, "
```

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}, $
db.accounts.update({name: t.destination, pendingTransactions: {$ne: t._id}}, {$inc: {balance: t.value
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"
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions"
```

Set Transaction State to Committed

Use the following `update()` operation to set the 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" : "comm
```

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"
{ "_id" : ObjectId("4d7bc984b8a04f5126961524"), "balance" : 1100, "name" : "B", "pendingTransactions"
```

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 the from 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 292)") but before the third step (i.e. "*applying the transaction to both accounts* (page 293).")

  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 292).")

- all failures that occur after the third step (i.e. "*applying the transaction to both accounts* (page 293)") but before the fifth step (i.e. "*setting the transaction state to done* (page 293).")

  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 293).")

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 293) (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 292) (i.e. the first step,) but before you *apply the transaction* (page 293) (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}, $pull: {p
db.accounts.update({name: t.destination, pendingTransactions: t._id}, {$inc: {balance: -t.value}, $pu
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 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 1 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 `A1`). If both applications begin processing the transaction which is still in the `initial` state (i.e. *step 1* (page 292)), 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 294), 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"
```

23.1.4 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 292) (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 293) (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.

23.2 Expire Data from Collections by Setting TTL

New in version 2.2. This document provides an introductions to MongoDB's "*time to live*" or "*TTL*" collection feature. Implemented as a special index type, TTL collections make it possible to store data in MongoDB and have the `mongod` automatically remove data after a specified period 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 period of time.

23.2.1 Background

Collections expire by way of a special index that keeps track of insertion time in conjunction with a background thread in `mongod` that regularly removes expired *documents* from the collection. You can use this feature to expire data from *replica sets* and *sharded clusters*.

Use the `expireAfterSeconds` option to the `ensureIndex` method in conjunction with a TTL value in seconds to create an expiring collection. TTL collections set the `usePowerOf2Sizes` collection flag, which means 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.

23.2.2 Constraints

Consider the following limitations:

- the indexed field must be a date *BSON type*. If the field does not have a date type, the data will not expire.
- you cannot create this index on the `_id` field, or a field that already has an index.
- the TTL index may not be compound (may not have multiple fields).
- 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.
- you cannot use a TTL index on a capped collection, because MongoDB cannot remove documents from a capped collection.

Note: TTL indexes expire data by removing documents in a background task that runs once a minute. As a result, the TTL index provides no guarantees that expired documents will not exist in the collection. Consider that:

- Documents may remain in a collection *after* they expire and before the background process runs.
- The duration of the removal operations depend on the workload of your `mongod` instance.

23.2.3 Enabling a TTL for a Collection

To set a TTL on the collection "`log.events`" for one hour use the following command at the `mongo` shell:

```
db.log.events.ensureIndex( { "status": 1 }, { expireAfterSeconds: 3600 } )
```

The `status` field *must* hold date/time information. MongoDB will automatically delete documents from this collection once the value of `status` is one or more hours old.

23.2.4 Replication

The background TTL thread *only* runs on *primaries*. All deletions operations replicate normally to the other members of the set.

Part X

Using the MongoDB Shell

See Also:

The introductory "Tutorial" in the MongoDB wiki and the "Mongo Shell" wiki pages for more information on the `mongo` shell.

CHAPTER

TWENTYFOUR

MONGO SHELL

- *mongo* (page 591)

CHAPTER

TWENTYFIVE

MONGODB SHELL INTERFACE

- reference/javascript
- reference/operators
- reference/commands
- *Aggregation Framework Reference* (page 269)
- reference/meta-query-operators

Part XI

Use Cases

The use case documents provide introductions to the patterns, design, and operation used in application development with MongoDB. Each document provides more concrete examples and implementation details to support core MongoDB use cases. These documents highlight application design, and data modeling strategies (*i.e. schema design*) for MongoDB with special attention to pragmatic considerations including indexing, performance, sharding, and scaling. Each document is distinct and can stand alone; however, each section builds on a set of common topics.

The *operational intelligence* case studies describe applications that collect machine generated data from logging systems, application output, and other systems. The *product data management* case studies address aspects of applications required for building product catalogs, and managing inventory in e-commerce systems. The *content management* case studies introduce basic patterns and techniques for building content management systems using MongoDB.

Finally, the *introductory application development tutorials with Python and MongoDB* (page 377), provides a complete and fully developed application that you can build using MongoDB and popular Python web development tool kits.

CHAPTER

TWENTYSIX

OPERATIONAL INTELLIGENCE

As an introduction to the use of MongoDB for operational intelligence and real time analytics use, "*Storing Log Data* (page 311)" document describes several ways and approaches to modeling and storing machine generated data with MongoDB. Then, "*Pre-Aggregated Reports* (page 321)" describes methods and strategies for processing data to generate aggregated reports from raw event-data. Finally "*Hierarchical Aggregation* (page 330)" presents a method for using MongoDB to process and store hierarchical reports (i.e. per-minute, per-hour, and per-day) from raw event data.

26.1 Storing Log Data

26.1.1 Overview

This document outlines the basic patterns and principles for using MongoDB as a persistent storage engine for log data from servers and other machine data.

Problem

Servers generate a large number of events (i.e. logging,) that contain useful information about their operation including errors, warnings, and users behavior. By default, most servers, store these data in plain text log files on their local file systems.

While plain-text logs are accessible and human-readable, they are difficult to use, reference, and analyze without holistic systems for aggregating and storing these data.

Solution

The solution described below assumes that each server generates events also consumes event data and that each server can access the MongoDB instance. Furthermore, this design assumes that the query rate for this logging data is substantially lower than common for logging applications with a high-bandwidth event stream.

Note: This case assumes that you're using an standard uncapped collection for this event data, unless otherwise noted. See the section on *capped collections* (page 321)

Schema Design

The schema for storing log data in MongoDB depends on the format of the event data that you're storing. For a simple example, consider standard request logs in the combined format from the Apache HTTP Server. A line from these logs may resemble the following:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.ex
```

The simplest approach to storing the log data would be putting the exact text of the log record into a document:

```
{
  _id: ObjectId('4f442120eb03305789000000'),
 line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http
}
```

While this solution is does capture all data in a format that MongoDB can use, the data is not particularly useful, or it's not terribly efficient: if you need to find events that the same page, you would need to use a regular expression query, which would require a full scan of the collection. The preferred approach is to extract the relevant information from the log data into individual fields in a MongoDB *document*.

When you extract data from the log into fields, pay attention to the data types you use to render the log data into MongoDB.

As you design this schema, be mindful that the data types you use to encode the data can have a significant impact on the performance and capability of the logging system. Consider the date field: In the above example, `[10/Oct/2000:13:55:36 -0700]` is 28 bytes long. If you store this with the UTC timestamp type, you can convey the same information in only 8 bytes.

Additionally, using proper types for your data also increases query flexibility: if you store date as a timestamp you can make date range queries, whereas it's very difficult to compare two *strings* that represent dates. The same issue holds for numeric fields; storing numbers as strings requires more space and is difficult to query.

Consider the following document that captures all data from the above log entry:

```
{
     _id: ObjectId('4f442120eb03305789000000'),
     host: "127.0.0.1",
     logname: null,
     user: 'frank',
     time: ISODate("2000-10-10T20:55:36Z"),
     path: "/apache_pb.gif",
     request: "GET /apache_pb.gif HTTP/1.0",
     status: 200,
     response_size: 2326,
     referrer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

When extracting data from logs and designing a schema, also consider what information you can omit from your log tracking system. In most cases there's no need to track *all* data from an event log, and you can omit other fields. To continue the above example, here the most crucial information may be the host, time, path, user agent, and referrer, as in the following example document:

```
{
     _id: ObjectId('4f442120eb03305789000000'),
     host: "127.0.0.1",
     time:  ISODate("2000-10-10T20:55:36Z"),
     path: "/apache_pb.gif",
     referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
```

```
    user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

You may also consider omitting explicit time fields, because the `ObjectId` embeds creation time:

```
{
    _id: ObjectId('4f442120eb03305789000000'),
    host: "127.0.0.1",
    path: "/apache_pb.gif",
    referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
    user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

System Architecture

The primary performance concern for event logging systems are:

1. how many inserts per second can it support, which limits the event throughput, and
2. how will the system manage the growth of event data, particularly concerning a growth in insert activity.

   In most cases the best way to increase the capacity of the system is to use an architecture with some sort of *partitioning* or *sharding* that distributes writes among a cluster of systems.

26.1.2 Operations

Insertion speed is the primary performance concern for an event logging system. At the same time, the system must be able to support flexible queries so that you can return data from the system efficiently. This section describes procedures for both document insertion and basic analytics queries.

The examples that follow use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Inserting a Log Record

Write Concern

MongoDB has a configurable *write concern*. This capability allows you to balance the importance of guaranteeing that all writes are fully recorded in the database with the speed of the insert.

For example, if you issue writes to MongoDB and do not require that the database issue any response, the write operations will return *very* fast (i.e. asynchronously,) but you cannot be certain that all writes succeeded. Conversely, if you require that MongoDB acknowledge every write operation, the database will not return as quickly but you can be certain that every item will be present in the database.

The proper write concern is often an application specific decision, and depends on the reporting requirements and uses of your analytics application.

Insert Performance

The following example contains the setup for a Python console session using PyMongo, with an event from the Apache Log:

```
>>> import bson
>>> import pymongo
>>> from datetime import datetime
>>> conn = pymongo.Connection()
>>> db = conn.event_db
>>> event = {
...     _id: bson.ObjectId(),
...     host: "127.0.0.1",
...     time:  datetime(2000,10,10,20,55,36),
...     path: "/apache_pb.gif",
...     referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
...     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
...}
```

The following command will insert the `event` object into the `events` collection.

```
>>> db.events.insert(event, safe=False)
```

By setting `safe=False`, you do not require that MongoDB acknowledges receipt of the insert. Although very fast, this is risky because the application cannot detect network and server failures.

If you want to ensure that MongoDB acknowledges inserts, you can pass `safe=True` argument as follows:

```
>>> db.events.insert(event, safe=True)
```

MongoDB also supports a more stringent level of write concern, if you have a lower tolerance for data loss:

You can ensure that MongoDB not only *acknowledge* receipt of the message but also commit the write operation to the on-disk journal before returning successfully to the application, use can use the following `insert()` operation:

```
>>> db.events.insert(event, j=True)
```

Note: `j=True` implies `safe=True`.

Finally, if you have *extremely low* tolerance for event data loss, you can require that MongoDB replicate the data to multiple *secondary replica set* members before returning:

```
>>> db.events.insert(event, w=2)
```

This will force your application to acknowledge that the data has replicated to 2 members of the *replica set*. You can combine options as well:

```
>>> db.events.insert(event, j=True, w=2)
```

In this case, your application will wait for a successful journal commit *and* a replication acknowledgment. This is the safest option presented in this section, but it is the slowest. There is always a trade-off between safety and speed.

Note: If possible, consider using bulk inserts to insert event data.

All write concern options apply to bulk inserts, but you can pass multiple events to the `insert()` method at once. Batch inserts allow MongoDB to distribute the performance penalty incurred by more stringent write concern across a group of inserts.

See Also:

"*Write Concern for Replica Sets* (page 54)" and `getLastError`.

Finding All Events for a Particular Page

The value in maintaining a collection of event data derives from being able to query that data to answer specific questions. You may have a number of simple queries that you may use to analyze these data.

As an example, you may want to return all of the events associated with specific value of a field. Extending the Apache access log example from above, a common case would be to query for all events with a specific value in the `path` field: This section contains a pattern for returning data and optimizing this operation.

Query

Use a query that resembles the following to return all documents with the `http://docs.mongodb.org/manual/apache_pb.gif` value in the `path` field:

```
>>> q_events = db.events.find({'path': '/apache_pb.gif'})
```

Note: If you choose to *shard* the collection that stores this data, the *shard key* you choose can impact the performance of this query. See the *sharding* (page 319) section of the sharding document.

Index Support

Adding an index on the `path` field would significantly enhance the performance of this operation.

```
>>> db.events.ensure_index('path')
```

Because the values of the `path` likely have a random distribution, in order to operate efficiently, the entire index should be resident in RAM. In this case, the number of distinct paths is typically small in relation to the number of documents, which will limit the space that the index requires.

If your system has a limited amount of RAM, or your data set has a wider distribution in values, you may need to re investigate your indexing support. In most cases, however, this index is entirely sufficient.

See Also:

The `db.collection.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Finding All the Events for a Particular Date

The next example describes the process for returning all the events for a particular date.

Query

To retrieve this data, use the following query:

```
>>> q_events = db.events.find('time':
...      { '$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)})
```

Index Support

In this case, an index on the `time` field would optimize performance:

```
>>> db.events.ensure_index('time')
```

Because your application is inserting events in order, the parts of the index that capture recent events will always be in active RAM. As a result, if you query primarily on recent data, MongoDB will be able to maintain a large index, quickly fulfill queries, and avoid using much system memory.

See Also:

The `db.events.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Finding All Events for a Particular Host/Date

The following example describes a more complex query for returning all events in the collection for a particular host on a particular date. This kinds analysis may be useful for investigating suspicious behavior by a specific user.

Query

Use a query that resembles the following:

```
>>> q_events = db.events.find({
...     'host': '127.0.0.1',
...     'time': {'$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)}
... })
```

This query selects *documents* from the `events` collection where the `host` field is `127.0.0.1` (i.e. local host), and the value of the `time` field represents a date that is on or after (i.e. `$gte`) `2000-10-10` but before (i.e. `$lt`) `2000-10-11`.

Index Support

The indexes you use may have significant implications for the performance of these kinds of queries. For instance, you *can* create a compound index on the `time` and `host` field, using the following command:

```
>>> db.events.ensure_index([('time', 1), ('host', 1)])
```

To analyze the performance for the above query using this index, issue the `q_events.explain()` method in a Python console. This will return something that resembles:

```
{ ...
  u'cursor': u'BtreeCursor time_1_host_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [
      [ datetime.datetime(2000, 10, 10, 0, 0),
        datetime.datetime(2000, 10, 11, 0, 0)]]
  },
  ...
  u'millis': 4,
  u'n': 11,
  u'nscanned': 1296,
  u'nscannedObjects': 11,
  ... }
```

This query had to scan 1296 items from the index to return 11 objects in 4 milliseconds. Conversely, you can test a different compound index with the `host` field first, followed by the `time` field. Create this index using the following operation:

```
>>> db.events.ensure_index([('host', 1), ('time', 1)])
```

Use the `q_events.explain()` operation to test the performance:

```
{ ...
  u'cursor': u'BtreeCursor host_1_time_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [[datetime.datetime(2000, 10, 10, 0, 0),
      datetime.datetime(2000, 10, 11, 0, 0)]]},
  ...
  u'millis': 0,
  u'n': 11,
  ...
  u'nscanned': 11,
  u'nscannedObjects': 11,
  ...
}
```

Here, the query had to scan 11 items from the index before returning 11 objects in less than a millisecond. By placing the more selective element of your query *first* in a compound index you may be able to build more useful queries.

Note: Although the index order has an impact query performance, remember that index scans are *much* faster than collection scans, and depending on your other queries, it may make more sense to use the `{ time: 1, host: 1 }` index depending on usage profile.

See Also:

The `db.events.ensureIndex()` JavaScript method and the `db.events.ensure_index()` method in PyMongo.

Counting Requests by Day and Page

The following example describes the process for using the collection of Apache access events to determine the number of request per resource (i.e. page) per day in the last month.

Aggregation

New in version 2.1. The *aggregation framework* provides the capacity for queries that select, process, and aggregate results from large numbers of documents. The `aggregate()` (and `aggregate` *command*) offers greater flexibility, capacity with less complexity than the existing `mapReduce` and `group` aggregation.

Consider the following aggregation *pipeline*: [1]

```
>>> result = db.command('aggregate', 'events', pipeline=[
...         {  '$match': {
...               'time': {
...                   '$gte': datetime(2000,10,1),
...                   '$lt':  datetime(2000,11,1) } } },
...         {  '$project': {
```

[1] To translate statements from the *aggregation framework* (page 253) to SQL, you can consider the `$match` (page 475) equivalent to `WHERE`, `$project` (page 477) to `SELECT`, and `$group` (page 473) to `GROUP BY`.

```
...                 'path': 1,
...                 'date': {
...                     'y': { '$year': '$time' },
...                     'm': { '$month': '$time' },
...                     'd': { '$dayOfMonth': '$time' } } } },
...         { '$group': {
...                 '_id': {
...                     'p':'$path',
...                     'y': '$date.y',
...                     'm': '$date.m',
...                     'd': '$date.d' },
...                 'hits': { '$sum': 1 } } },
...         ])
```

This command aggregates documents from the `events` collection with a pipeline that:

1. Uses the `$match` (page 475) to limit the documents that the aggregation framework must process. `$match` (page 475) is similar to a `find()` query.

   This operation selects all documents where the value of the `time` field represents a date that is on or after (i.e. `$gte`) `2000-10-10` but before (i.e. `$lt`) `2000-10-11`.

2. Uses the `$project` (page 477) to limit the data that continues through the pipeline. This operator:

   - Selects the `path` field.
   - Creates a `y` field to hold the year, computed from the `time` field in the original documents.
   - Creates a `m` field to hold the month, computed from the `time` field in the original documents
   - Creates a `d` field to hold the day, computed from the `time` field in the original documents.

3. Uses the `$group` (page 473) to create new computed documents. This step will create a single new document for each unique path/date combination. The documents take the following form:

   - the `_id` field holds a sub-document with the contents `path` field from the original documents in the `p` field, with the `date` fields from the `$project` (page 477) as the remaining fields.
   - the `hits` field use the `$sum` (page 481) statement to increment a counter for every document in the group. In the aggregation output, this field holds the total number of documents at the beginning of the aggregation pipeline with this unique date and path.

Note: In sharded environments, the performance of aggregation operations depends on the *shard key*. Ideally, all the items in a particular `$group` (page 473) operation will reside on the same server.

While this distribution of documents would occur if you chose the `time` field as the shard key, a field like `path` also has this property and is a typical choice for sharding. Also see the "*sharding considerations* (page 319)." of this document for additional recommendations for using sharding.

See Also:

"*Aggregation Framework* (page 253)"

Index Support

To optimize the aggregation operation, ensure that the initial `$match` (page 475) query has an index. Use the following command to create an index on the `time` field in the `events` collection:

```
>>> db.events.ensure_index('time')
```

Note: If you have already created a compound index on the `time` and `host` (i.e. `{ time: 1, host, 1 }`,) MongoDB will use this index for range queries on just the `time` field. Do not create an additional index, in these situations.

26.1.3 Sharding

Eventually your system's events will exceed the capacity of a single event logging database instance. In these situations you will want to use a *sharded cluster*, which takes advantage of MongoDB's *sharding* functionality. This section introduces the unique sharding concerns for this event logging case.

See Also:

"*FAQ: Sharding with MongoDB* (page 423)" and the "Sharding wiki page.

Limitations

In a sharded environment the limitations on the maximum insertion rate are:

- the number of shards in the cluster.
- the *shard key* you chose.

Because MongoDB distributed data in using "ranges" (i.e. *chunks*) of *keys*, the choice of shard key can control how MongoDB distributes data and the resulting systems' capacity for writes and queries.

Ideally, your shard key should allow insertions balance evenly among the shards [2] and for most queries to only *need* to access a single shard. [3] Continue reading for an analysis of a collection of shard key choices.

Shard by Time

While using the timestamp, or the `ObjectId` in the `_id` field, [4] would distribute your data evenly among shards, these keys lead to two problems:

1. All inserts always flow to the same shard, which means that your *sharded cluster* will have the same write throughput as a standalone instance.
2. Most reads will tend to cluster on the same shard, as analytics queries.

Shard by a Semi-Random Key

To distribute data more evenly among the shards, you may consider using a more "random" piece of data, such as a hash of the `_id` field (i.e. the `ObjectId` as a *shard key*.

While this introduces some additional complexity into your application, to generate the key, it will distribute writes among the shards. In these deployments having 5 shards will provide 5 times the write capacity as a single instance.

Using this shard key, or any hashed value as a key presents the following downsides:

- the shard key, and the index on the key will consume additional space in the database.

[2] For this reason, avoid shard keys based on the timestamp or the insertion time (i.e. the `ObjectId`) because all writes will end up on a single node.

[3] For this reason, avoid randomized shard keys (e.g. hash based shard keys) because any query will have to access all shards in the cluster.

[4] The `ObjectId` derives from the creation time, and is effectively a timestamp in this case.

- queries, unless they include the shard key itself, [5] must run in parallel on all shards, which may lead to degraded performance.

This might be an acceptable trade-off in some situations. The workload of event logging systems tends to be heavily skewed toward writing, read performance may not be as critical as more robust write performance.

Shard by an Evenly-Distributed Key in the Data Set

If a field in your documents has values that are evenly distributed among the documents, you may consider using this key as a *shard key*.

Continuing the example from above, you may consider using the `path` field. Which may have a couple of advantages:

1. writes will tend to balance evenly among shards.
2. reads will tend to be selective and local to a single shard if the query selects on the `path` field.

There are a few potential problems with these kinds of shard keys:

1. If a large number of documents will have the same shard key, you run the risk of having a portion of your data collection MongoDB cannot distribute throughout the cluster.
2. If there are a small number of possible values, there may be a limit to how much MongoDB will be able to distribute the data among the shard.

Note: Test using your existing data to ensure that the distribution is truly even, and that there is a sufficient quantity of distinct values for the shard key.

Shard by Combine a Natural and Synthetic Key

MongoDB supports compound *shard keys* that combine the best aspects of *sharding by a evenly distributed key in the set* (page 320) and *sharding by a random key* (page 319). In these situations, the shard key would resemble `{ path: 1 , ssk: 1 }` where, `path` is an often used "natural key, or value from your data and `ssk` is a hash of the `_id` field. [6]

Using this type of shard key, data is largely distributed by the natural key, or `path`, which makes most queries that access the `path` field local to a single shard or group of shards. At the same time, if there is not sufficient distribution for specific values of `path`, the `ssk` makes it possible for MongoDB to create *chunks* and data across the cluster.

In most situations, these kinds of keys provide the ideal balance between distributing writes across the cluster and ensuring that most queries will only need to access a select number of shards.

Test with Your Own Data

Selecting shard keys is difficult because: there are no definitive "best-practices," the decision has a large impact on performance, and it is difficult or impossible to change the shard key after making the selection.

The *sharding options* (page 319) provides a good starting point for thinking about *shard key* selection. Nevertheless, the best way to select a shard key is to analyze the actual insertions and queries from your own application.

[5] Typically, it is difficult to use these kinds of shard keys in queries.
[6] You must still calculate the value of this synthetic key in your application when you insert documents into your collection.

26.1.4 Managing Event Data Growth

Without some strategy for managing the size of your database, most event logging systems can grow infinitely. This is particularly important in the context of MongoDB may not relinquish data to the file system in the way you might expect. Consider the following strategies for managing data growth:

Capped Collections

Depending on your data retention requirements as well as your reporting and analytics needs, you may consider using a *capped collection* to store your events. Capped collections have a fixed size, and drop old data when inserting new data after reaching cap.

Note: In the current version, it is not possible to shard capped collections.

Multiple Collections, Single Database

Strategy: Periodically rename your event collection so that your data collection rotates in much the same way that you might rotate log files. When needed, you can drop the oldest collection from the database.

This approach has several advantages over the single collection approach:

1. Collection renames are fast and atomic.
2. MongoDB does not bring any document into memory to drop a collection.
3. MongoDB can effectively reuse space freed by removing entire collections without leading to data fragmentation.

Nevertheless, this operation may increase some complexity for queries, if any of your analyses depend on events that may reside in the current and previous collection. For most real time data collection systems, this approach is the most ideal.

Multiple Databases

Strategy: Rotate databases rather than collections, as in the "*Multiple Collections, Single Database* (page 321) example.

While this *significantly* increases application complexity for insertions and queries, when you drop old databases, MongoDB will return disk space to the file system. This approach makes the most sense in scenarios where your event insertion rates and/or your data retention rates were extremely variable.

For example, if you are performing a large backfill of event data and want to make sure that the entire set of event data for 90 days is available during the backfill, during normal operations you only need 30 days of event data, you might consider using multiple databases.

26.2 Pre-Aggregated Reports

26.2.1 Overview

This document outlines the basic patterns and principles for using MongoDB as an engine for collecting and processing events in real time for use in generating up to the minute or second reports.

Problem

Servers and other systems can generate a large number of documents, and it can be difficult to access and analyze such large collections of data originating from multiple servers.

This document makes the following assumptions about real-time analytics:

- There is no need to retain transactional event data in MongoDB, and how your application handles transactions is outside of the scope of this document.
- You require up-to-the minute data, or up-to-the-second if possible.
- The queries for ranges of data (by time) must be as fast as possible.

See Also:

"*Storing Log Data* (page 311)."

Solution

The solution described below assumes a simple scenario using data from web server access logs. With this data, you will want to return the number of hits to a collection of web sites at various levels of granularity based on time (i.e. by minute, hour, day, week, and month) as well as by the path of a resource.

To achieve the required performance to support these tasks, *upserts* and `increment` operations will allow you to calculate statistics, produce simple range-based queries, and generate filters to support time-series charts of aggregated data.

26.2.2 Schema

Schemas for real-time analytics systems must support simple and fast query and update operations. In particular, attempt to avoid the following situations which can degrade performance:

- *documents* growing significantly after creation.

  Document growth forces MongoDB to move the document on disk, which can be time and resource consuming relative to other operations;

- queries requiring MongoDB to scan documents in the collection without using indexes; and
- deeply nested documents that make accessing particular fields slow.

Intuitively, you may consider keeping "hit counts" in individual documents with one document for every unit of time (i.e. minute, hour, day, etc.) However, queries must return multiple documents for all non-trivial time-rage queries, which can slow overall query performance.

Preferably, to maximize query performance, use more complex documents, and keep several aggregate values in each document. The remainder of this section outlines several schema designs that you may consider for this real-time analytics system. While there is no single pattern for every problem, each pattern is more well suited to specific classes of problems.

One Document Per Page Per Day

Consider the following example schema for a solution that stores all statistics for a single day and page in a single *document*:

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: 5468426,
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": 3612,
        "1": 3241,
        ...
        "1439": 2819 }
}
```

This approach has a couple of advantages:

- For every request on the website, you only need to update one document.
- Reports for time periods within the day, for a single page require fetching a single document.

There are, however, significant issues with this approach. The most significant issue is that, as you *upsert* data into the `hourly` and `monthly` fields, the document grows. Although MongoDB will pad the space allocated to documents, it must still will need to reallocate these documents multiple times throughout the day, which impacts performance.

Pre-allocate Documents

Simple Pre-Allocation

To mitigate the impact of repeated document migrations throughout the day, you can tweak the "*one document per page per day* (page 322)" approach by adding a process that "pre-allocates" documents with fields that hold `0` values throughout the previous day. Thus, at midnight, new documents will exist.

Note: To avoid situations where your application must pre-allocate large numbers of documents at midnight, it's best to create documents throughout the previous day by *upserting* randomly when you update a value in the current day's data.

This requires some tuning, to balance two requirements:

1. your application should have pre-allocated all or nearly all of documents by the end of the day.
2. your application should infrequently pre-allocate a document that already exists to save time and resources on extraneous upserts.

As a starting point, consider the average number of hits a day (`h`), and then upsert a blank document upon update with a probability of `1/h`.

Pre-allocating increases performance by initializing all documents with `0` values in all fields. After create, documents will never grow. This means that:

1. there will be no need to migrate documents within the data store, which is a problem in the "*one document per page per day* (page 322)" approach.

2. MongoDB will not add padding to the records, which leads to a more compact data representation and better memory use of your memory.

Add Intra-Document Hierarchy

Note: MongoDB stores *BSON documents* as a sequence of fields and values, *not* as a hash table. As a result, writing to the field `stats.mn.0` is considerably faster than writing to `stats.mn.1439`.

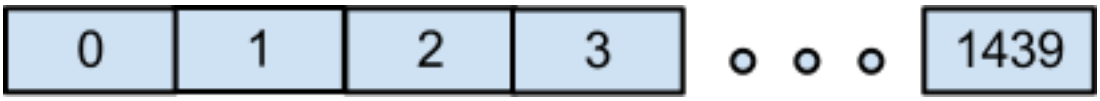


Figure 26.1: In order to update the value in minute #1349, MongoDB must skip over all 1349 entries before it.

To optimize update and insert operations you can introduce intra-document hierarchy. In particular, you can split the `minute` field up into 24 hourly fields:

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: 5468426,
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": {
            "0": 3612,
            "1": 3241,
            ...
            "59": 2130 },
        "1": {
        "60": ... ,
        },
        ...
        "23": {
            ...
            "1439": 2819 }
    }
}
```

This allows MongoDB to "skip forward" throughout the day when updating the minute data, which makes the update performance more uniform and faster later in the day.

Separate Documents by Granularity Level

Pre-allocating documents (page 323) is a reasonable design for storing intra-day data, but the model breaks down when displaying data over longer multi-day periods like months or quarters. In these cases, consider storing daily statistics in a single document as above, and then aggregate monthly data into a separate document.

Figure 26.2: To update the value in minute #1349, MongoDB first skips the first 23 hours and then skips 59 minutes for only 82 skips as opposed to 1439 skips in the previous schema.

This introduce a second set of *upsert* operations to the data collection and aggregation portion of your application but the gains reduction in disk seeks on the queries, should be worth the costs. Consider the following example schema:

1. Daily Statistics

```
{
    _id: "20101010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-10T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    hourly: {
        "0": 227850,
        "1": 210231,
        ...
        "23": 20457 },
    minute: {
        "0": {
            "0": 3612,
            "1": 3241,
            ...
            "59": 2130 },
        "1": {
            "0": ...,
        },
        ...
        "23": {
            "59": 2819 }
    }
}
```

2. Monthly Statistics

```
{
    _id: "201010/site-1/apache_pb.gif",
    metadata: {
        date: ISODate("2000-10-00T00:00:00Z"),
        site: "site-1",
        page: "/apache_pb.gif" },
    daily: {
        "1": 5445326,
        "2": 5214121,
        ... }
}
```

26.2.3 Operations

This section outlines a number of common operations for building and interacting with real-time-analytics reporting system. The major challenge is in balancing performance and write (i.e. *upsert*) performance. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Log an Event

Logging an event such as a page request (i.e. "hit") is the main "write" activity for your system. To maximize performance, you'll be doing in-place updates with the *upsert* operation. Consider the following example:

```
from datetime import datetime, time

def log_hit(db, dt_utc, site, page):

    # Update daily stats doc
    id_daily = dt_utc.strftime('%Y%m%d/') + site + page
    hour = dt_utc.hour
    minute = dt_utc.minute

    # Get a datetime that only includes date info
    d = datetime.combine(dt_utc.date(), time.min)
    query = {
        '_id': id_daily,
        'metadata': { 'date': d, 'site': site, 'page': page } }
    update = { '$inc': {
            'hourly.%d' % (hour,): 1,
            'minute.%d.%d' % (hour,minute): 1 } }
    db.stats.daily.update(query, update, upsert=True)

    # Update monthly stats document
    id_monthly = dt_utc.strftime('%Y%m/') + site + page
    day_of_month = dt_utc.day
    query = {
        '_id': id_monthly,
        'metadata': {
            'date': d.replace(day=1),
            'site': site,
            'page': page } }
    update = { '$inc': {
            'daily.%d' % day_of_month: 1} }
    db.stats.monthly.update(query, update, upsert=True)
```

The upsert operation (i.e. `upsert=True`) performs an update if the document exists, and an insert if the document does not exist.

Note: This application requires upserts, because the *pre-allocation* (page 327) method only pre-allocates new documents with a high probability, not with complete certainty.

Without preallocation, you end up with a dynamically growing document, slowing upserts as MongoDB moves documents to accommodate growth.

Pre-allocate

To prevent document growth, you can preallocate new documents before the system needs them. As you create new documents, set all values to `0` for so that documents will not grow to accommodate updates. Consider the following `preallocate()` function:

```
def preallocate(db, dt_utc, site, page):

    # Get id values
    id_daily = dt_utc.strftime('%Y%m%d/') + site + page
    id_monthly = dt_utc.strftime('%Y%m/') + site + page

    # Get daily metadata
    daily_metadata = {
        'date': datetime.combine(dt_utc.date(), time.min),
        'site': site,
        'page': page }
    # Get monthly metadata
    monthly_metadata = {
        'date': daily_m['d'].replace(day=1),
        'site': site,
        'page': page }

    # Initial zeros for statistics
    hourly = dict((str(i), 0) for i in range(24))
    minute = dict(
        (str(i), dict((str(j), 0) for j in range(60)))
        for i in range(24))
    daily = dict((str(i), 0) for i in range(1, 32))

    # Perform upserts, setting metadata
    db.stats.daily.update(
        {
            '_id': id_daily,
            'hourly': hourly,
            'minute': minute},
        { '$set': { 'metadata': daily_metadata }},
        upsert=True)
    db.stats.monthly.update(
        {
            '_id': id_monthly,
            'daily': daily },
        { '$set': { 'm': monthly_metadata }},
        upsert=True)
```

The function pre-allocated both the monthly *and* daily documents at the same time. The performance benefits from separating these operations are negligible, so it's reasonable to keep both operations in the same function.

Ideally, your application should pre-allocate documents *before* needing to write data to maintain consistent update performance. Additionally, its important to avoid causing a spike in activity and latency by creating documents all at once.

In the following example, document updates (i.e. "`log_hit()`") will also pre-allocate a document probabilistically. However, by "tuning probability," you can limit redundant `preallocate()` calls.

```
from random import random
from datetime import datetime, timedelta, time

# Example probability based on 500k hits per day per page
```

```
prob_preallocate = 1.0 / 500000

def log_hit(db, dt_utc, site, page):
    if random.random() < prob_preallocate:
        preallocate(db, dt_utc + timedelta(days=1), site_page)
    # Update daily stats doc
    ...
```

Using this method, there will be a high probability that each document will already exist before your application needs to issue update operations. You'll also be able to prevent a regular spike in activity for pre-allocation, and be able to eliminate document growth.

Retrieving Data for a Real-Time Chart

This example describes fetching the data from the above MongoDB system, for use in generating a chart that displays the number of hits to a particular resource over the last hour.

Querying

Use the following query in a `find_one` operation at the Python/PyMongo console to retrieve the number of hits to a specific resource (i.e. `http://docs.mongodb.org/manual/index.html`) with minute-level granularity:

```
>>``db.stats.daily.find_one(
...     {'metadata': {'date':dt, 'site':'site-1', 'page':'/index.html'}},
...     { 'minute': 1 })
```

Use the following query to retrieve the number of hits to a resource over the last day, with hour-level granularity:

```
>>> db.stats.daily.find_one(
...     {'metadata': {'date':dt, 'site':'site-1', 'page':'/foo.gif'}},
...     { 'hy': 1 })
```

If you want a few days of hourly data, you can use a query in the following form:

```
>>> db.stats.daily.find(
...     {
...         'metadata.date': { '$gte': dt1, '$lte': dt2 },
...         'metadata.site': 'site-1',
...         'metadata.page': '/index.html'},
...     { 'metadata.date': 1, 'hourly': 1 } },
...     sort=[('metadata.date', 1)])
```

Indexing

To support these query operation, create a compound index on the following daily statistics fields: `metadata.site`, `metadata.page`, and `metadata.date` (in that order.) Use the following operation at the Python/PyMongo console.

```
>>> db.stats.daily.ensure_index([
...     ('metadata.site', 1),
...     ('metadata.page', 1),
...     ('metadata.date', 1)])
```

This index makes it possible to efficiently run the query for multiple days of hourly data. At the same time, any compound index on page and date, will allow you to query efficiently for a single day's statistics.

Get Data for a Historical Chart

Querying

To retrieve daily data for a single month, use the following query:

```
>>> db.stats.monthly.find_one(
...     {'metadata':
...         {'date':dt,
...         'site': 'site-1',
...         'page':'/index.html'}},
...     { 'daily': 1 })
```

To retrieve several months of daily data, use a variation on the above query:

```
>>> db.stats.monthly.find(
...     {
...         'metadata.date': { '$gte': dt1, '$lte': dt2 },
...         'metadata.site': 'site-1',
...         'metadata.page': '/index.html'},
...     { 'metadata.date': 1, 'daily': 1 } },
...     sort=[('metadata.date', 1)])
```

Indexing

Create the following index to support these queries for monthly data on the `metadata.site`, `metadata.page`, and `metadata.date` fields:

```
>>> db.stats.monthly.ensure_index([
...     ('metadata.site', 1),
...     ('metadata.page', 1),
...     ('metadata.date', 1)])
```

This field order will efficiently support range queries for a single page over several months.

26.2.4 Sharding

The only potential limits on the performance of this system are the number of *shards* in your *system*, and the *shard key* that you use.

> An ideal shard key will distribute *upserts* between the shards while routing all queries to a single shard, or a small number of shards.

While your choice of shard key may depend on the precise workload of your deployment, consider using `{ metadata.site: 1, metadata.page: 1 }` as a *shard key*. The combination of site and page (or event) will lead to a well balanced cluster for most deployments.

Enable sharding for the daily statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.daily', {
...     key : { 'metadata.site': 1, 'metadata.page' : 1 } })
```

Upon success, you will see the following response:

```
{ "collectionsharded" : "stats.daily", "ok" : 1 }
```

Enable sharding for the monthly statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.monthly', {
...     key : { 'metadata.site': 1, 'metadata.page' : 1 } })
```

Upon success, you will see the following response:

```
{ "collectionsharded" : "stats.monthly", "ok" : 1 }
```

One downside of the `{ metadata.site: 1, metadata.page: 1 }` *shard key* is: if one page dominates all your traffic, all updates to that page will go to a single shard. This is basically unavoidable, since all update for a single page are going to a single *document*.

You may wish to include the date in addition to the site, and page fields so that MongoDB can split histories so that you can serve different historical ranges with different shards. Use the following `shardCollection` command to shard the daily statistics collection in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.daily', {
...     'key':{'metadata.site':1,'metadata.page':1,'metadata.date':1}})
{ "collectionsharded" : "stats.daily", "ok" : 1 }
```

Enable sharding for the monthly statistics collection with the following `shardCollection` command in the Python/PyMongo console:

```
>>> db.command('shardCollection', 'stats.monthly', {
...     'key':{'metadata.site':1,'metadata.page':1,'metadata.date':1}})
{ "collectionsharded" : "stats.monthly", "ok" : 1 }
```

Note: Determine your actual requirements and load before deciding to shard. In many situations a single MongoDB instance may be able to keep track of all events and pages.

26.3 Hierarchical Aggregation

26.3.1 Overview

Background

If you collect a large amount of data, but do not *pre-aggregate* (page 321), and you want to have access to aggregated information and reports, then you need a method to aggregate these data into a usable form. This document provides an overview of these aggregation patterns and processes.

For clarity, this case study assumes that the incoming event data resides in a collection named `events`. For details on how you might get the event data into the events collection, please see "*Storing Log Data* (page 311)" document. This document continues using this example.

Solution

The first step in the aggregation process is to aggregate event data into the finest required granularity. Then use this aggregation to generate the next least specific level granularity and this repeat process until you have generated all required views.

The solution uses several collections: the raw data (i.e. `events`) collection as well as collections for aggregated hourly, daily, weekly, monthly, and yearly statistics. All aggregations use the `mapReduce` *command*, in a hierarchical process. The following figure illustrates the input and output of each job:

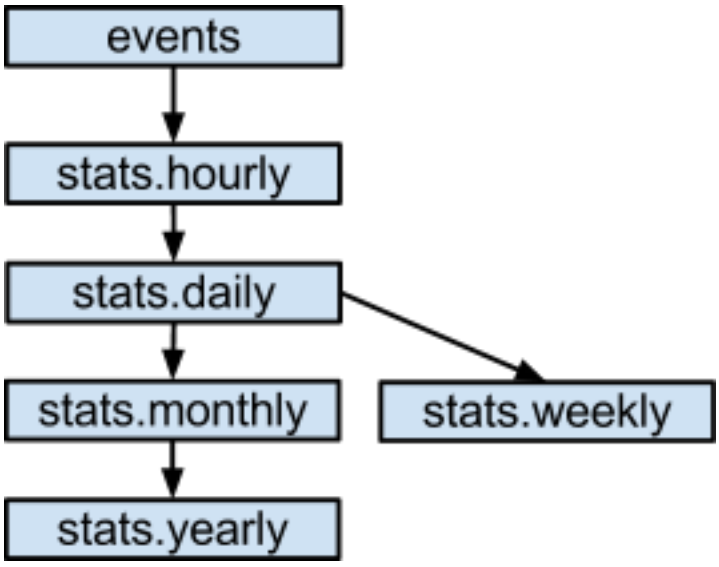


Figure 26.3: Hierarchy of data aggregation.

Note: Aggregating raw events into an hourly collection is qualitatively different from the operation that aggregates hourly statistics into the daily collection.

See Also:

Map-reduce and the MapReduce wiki page for more information on the Map-reduce data aggregation paradigm.

26.3.2 Schema

When designing the schema for event storage, it's important to track the events included in the aggregation and events that are not yet included.

Relational Approach

A simple tactic from relational database, uses an auto-incremented integer as the primary key. However, this introduces a significant performance penalty for event logging process because the aggregation process must fetch new keys one at a time.

If you can batch your inserts into the `events` collection, you can use an auto-increment primary key by using the `find_and_modify` command to generate the `_id` values, as in the following example:

```
>>> obj = db.my_sequence.find_and_modify(
...     query={'_id':0},
...     update={'$inc': {'inc': 50}}
...     upsert=True,
...     new=True)
>>> batch_of_ids = range(obj['inc']-50, obj['inc'])
```

However, in most cases you can simply include a timestamp with each event that you can use to distinguish processed events from unprocessed events.

This example assumes that you are calculating average session length for logged-in users on a website. The events will have the following form:

```
{
    "userid": "rick",
    "ts": ISODate('2010-10-10T14:17:22Z'),
    "length":95
}
```

The operations described in the next session will calculate total and average session times for each user at the hour, day, week, month and year. For each aggregation you will want to store the number of sessions so that MongoDB can incrementally recompute the average session times. The aggregate document will resemble the following:

```
{
   _id: { u: "rick", d: ISODate("2010-10-10T14:00:00Z") },
   value: {
       ts: ISODate('2010-10-10T15:01:00Z'),
       total: 254,
       count: 10,
       mean: 25.4 }
}
```

Note: The timestamp value in the `_id` sub-document, which will allow you to incrementally update documents at various levels of the hierarchy.

26.3.3 Operations

This section assumes that all events exist in the `events` collection and have a timestamp. The operations, thus are to aggregate from the `events` collection into the smallest aggregate–hourly totals– and then aggregate from the hourly totals into coarser granularity levels. In all cases, these operations will store aggregation time as a `last_run` variable.

Creating Hourly Views from Event Collections

Aggregation

Note: Although this solution uses Python and PyMongo to connect with MongoDB, you must pass JavaScript functions (i.e. `mapf`, `reducef`, and `finalizef`) to the `mapReduce` command.

Begin by creating a map function, as below:

```
mapf_hour = bson.Code('''function() {
    var key = {
        u: this.userid,
        d: new Date(
            this.ts.getFullYear(),
            this.ts.getMonth(),
            this.ts.getDate(),
            this.ts.getHours(),
            0, 0, 0);
    emit(
```

```
        key,
        {
            total: this.length,
            count: 1,
            mean: 0,
            ts: new Date(); });
}''')
```

In this case, it emits key-value pairs that contain the data you want to aggregate as you'd expect. The function also emits a `ts` value that makes it possible to cascade aggregations to coarser grained aggregations (i.e. hour to day, etc.)

Consider the following reduce function:

```
reducef = bson.Code('''function(key, values) {
    var r = { total: 0, count: 0, mean: 0, ts: null };
    values.forEach(function(v) {
        r.total += v.total;
        r.count += v.count;
    });
    return r;
}''')
```

The reduce function returns a document in the same format as the output of the map function. This pattern for map and reduce functions makes map-reduce processes easier to test and debug.

While the reduce function ignores the `mean` and `ts` (timestamp) values, the finalize step, as follows, computes these data:

```
finalizef = bson.Code('''function(key, value) {
    if(value.count > 0) {
       value.mean = value.total / value.count;
    }
    value.ts = new Date();
    return value;
}''')
```

With the above function the `map_reduce` operation itself will resemble the following:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)
query = { 'ts': { '$gt': last_run, '$lt': cutoff } }

db.events.map_reduce(
    map=mapf_hour,
    reduce=reducef,
    finalize=finalizef,
    query=query,
    out={ 'reduce': 'stats.hourly' })

last_run = cutoff
```

The `cuttoff` variable allows you to process all events that have occurred since the last run but before 1 minute ago. This allows for some delay in logging events. You can safely run this aggregation as often as you like, provided that you update the `last_run` variable each time.

Indexing

Create an index on the timestamp (i.e. the `ts` field) to support the query selection of the `map_reduce` operation. Use the following operation at the Python/PyMongo console:

```
>>> db.events.ensure_index('ts')
```

Deriving Day-Level Data

Aggregation

To calculate daily statistics, use the hourly statistics as input. Begin with the following map function:

```
mapf_day = bson.Code('''function() {
    var key = {
        u: this._id.u,
        d: new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            this._id.d.getDate(),
            0, 0, 0, 0) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

The map function for deriving day-level data differs from the initial aggregation above in the following ways:

- the aggregation key is the (userid, date) rather than (userid, hour) to support daily aggregation.
- the keys and values emitted (i.e. `emit()`) are actually the total and count values from the hourly aggregates rather than properties from event documents.

  This is the case for all the higher-level aggregation operations.

Because the output of this map function is the same as the previous map function, you can use the same reduce and finalize functions.

The actual code driving this level of aggregation is as follows:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)
query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }

db.stats.hourly.map_reduce(
    map=mapf_day,
    reduce=reducef,
    finalize=finalizef,
    query=query,
    out={ 'reduce': 'stats.daily' })

last_run = cutoff
```

There are a couple of things to note here. First of all, the query is not on `ts` now, but `value.ts`, the timestamp written during the finalization of the hourly aggregates. Also note that you are, in fact, aggregating from the `stats.hourly` collection into the `stats.daily` collection.

Indexing

Because you will run the query option regularly which finds on the `value.ts` field, you may wish to create an index to support this. Use the following operation in the Python/PyMongo shell to create this index:

```
>>> db.stats.hourly.ensure_index('value.ts')
```

Weekly and Monthly Aggregation

Aggregation

You can use the aggregated day-level data to generate weekly and monthly statistics. A map function for generating weekly data follows:

```
mapf_week = bson.Code('''function() {
    var key = {
        u: this._id.u,
        d: new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

Here, to get the group key, the function takes the current and subtracts days until you get the beginning of the week. In the weekly map function, you'll use the first day of the month as the group key, as follows:

```
mapf_month = bson.Code('''function() {
        d: new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0) };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}''')
```

These map functions are identical to each other except for the date calculation.

Indexing

Create additional indexes to support the weekly and monthly aggregation options on the `value.ts` field. Use the following operation in the Python/PyMongo shell.

```
>>> db.stats.daily.ensure_index('value.ts')
>>> db.stats.monthly.ensure_index('value.ts')
```

Refactor Map Functions

Use Python's string interpolation to refactor the map function definitions as follows:

```
mapf_hierarchical = '''function() {
    var key = {
        u: this._id.u,
        d: %s };
    emit(
        key,
        {
            total: this.value.total,
            count: this.value.count,
            mean: 0,
            ts: null });
}'''

mapf_day = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            this._id.d.getDate(),
            0, 0, 0, 0)''')

mapf_week = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.valueOf()
            - dt.getDay()*24*60*60*1000)''')

mapf_month = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            this._id.d.getMonth(),
            1, 0, 0, 0, 0)''')

mapf_year = bson.Code(
    mapf_hierarchical % '''new Date(
            this._id.d.getFullYear(),
            1, 1, 0, 0, 0, 0)''')
```

You can create a `h_aggregate` function to wrap the `map_reduce` operation, as below, to reduce code duplication:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
     query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
     icollection.map_reduce(
         map=mapf,
         reduce=reducef,
         finalize=finalizef,
         query=query,
         out={ 'reduce': ocollection.name })
```

With `h_aggregate` defined, you can perform all aggregation operations as follows:

```
cutoff = datetime.utcnow() - timedelta(seconds=60)

h_aggregate(db.events, db.stats.hourly, mapf_hour, cutoff, last_run)
h_aggregate(db.stats.hourly, db.stats.daily, mapf_day, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.weekly, mapf_week, cutoff, last_run)
h_aggregate(db.stats.daily, db.stats.monthly, mapf_month, cutoff, last_run)
```

```
h_aggregate(db.stats.monthly, db.stats.yearly, mapf_year, cutoff, last_run)

last_run = cutoff
```

As long as you save and restore the `last_run` variable between aggregations, you can run these aggregations as often as you like since each aggregation operation is incremental.

26.3.4 Sharding

Ensure that you choose a *shard key* that is not the incoming timestamp, but rather something that varies significantly in the most recent documents. In the example above, consider using the `userid` as the most significant part of the shard key.

To prevent a single, active user from creating a large, *chunk* that MongoDB cannot split, use a compound shard key with (username, timestamp) on the events collection. Consider the following:

```
>>> db.command('shardCollection','events', {
... 'key' : { 'userid': 1, 'ts' : 1} } )
{ "collectionsharded": "events", "ok" : 1 }
```

To shard the aggregated collections you must use the `_id` field, so you can issue the following group of shard operations in the Python/PyMongo shell:

```
db.command('shardCollection', 'stats.daily', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.weekly', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.monthly', {
     'key': { '_id': 1 } })
db.command('shardCollection', 'stats.yearly', {
     'key': { '_id': 1 } })
```

You should also update the `h_aggregate` map-reduce wrapper to support sharded output Add `'sharded':True` to the `out` argument. See the full sharded `h_aggregate` function:

```
def h_aggregate(icollection, ocollection, mapf, cutoff, last_run):
     query = { 'value.ts': { '$gt': last_run, '$lt': cutoff } }
     icollection.map_reduce(
         map=mapf,
         reduce=reducef,
         finalize=finalizef,
         query=query,
         out={ 'reduce': ocollection.name, 'sharded': True })
```

CHAPTER

TWENTYSEVEN

PRODUCT DATA MANAGEMENT

MongoDB's flexible schema makes it particularly well suited to storing information for product data management and e-commerce websites and solutions. The "*Product Catalog* (page 339)" document describes methods and practices for modeling and managing a product catalog using MongoDB, while the "*Inventory Management* (page 346)" document introduces a pattern for handling interactions between inventory and users' shopping carts. Finally the "*Category Hierarchy* (page 353)" document describes methods for interacting with category hierarchies in MongoDB.

27.1 Product Catalog

27.1.1 Overview

This document describes the basic patterns and principles for designing an E-Commerce product catalog system using MongoDB as a storage engine.

Problem

Product catalogs must have the capacity to store many differed types of objects with different sets of attributes. These kinds of data collections are quite compatible with MongoDB's data model, but many important considerations and design decisions remain.

Solution

For relational databases, there are several solutions that address this problem, each with a different performance profile. This section examines several of these options and then describes the preferred MongoDB solution.

SQL and Relational Data Models

Concrete Table Inheritance

One approach, in a relational model, is to create a table for each product category. Consider the following example SQL statement for creating database tables:

```
CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
```

```
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
...
```

This approach has limited flexibility for two key reasons:

- You must create a new table for every new category of products.
- You must explicitly tailor all queries for the exact type of product.

Single Table Inheritance

Another relational data model uses a single table for all product categories and adds new columns anytime you need to store data regarding a new type of product. Consider the following SQL statement:

```
CREATE TABLE `product` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
    `genre_1` varchar(255) DEFAULT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`))
```

This approach is more flexible than concrete table inheritance: it allows single queries to span different product types, but at the expense of space.

Multiple Table Inheritance

Also in the relational model, you may use a "multiple table inheritance" pattern to represent common attributes in a generic "product" table, with some variations in individual category product tables. Consider the following SQL statement:

```
CREATE TABLE `product` (
    `sku` char(8) NOT NULL,
    `title` varchar(255) DEFAULT NULL,
    `description` varchar(255) DEFAULT NULL,
    `price`, ...
    PRIMARY KEY(`sku`))

CREATE TABLE `product_audio_album` (
    `sku` char(8) NOT NULL,
    ...
    `artist` varchar(255) DEFAULT NULL,
    `genre_0` varchar(255) DEFAULT NULL,
```

```
    `genre_1` varchar(255) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
CREATE TABLE `product_film` (
    `sku` char(8) NOT NULL,
    ...
    `title` varchar(255) DEFAULT NULL,
    `rating` char(8) DEFAULT NULL,
    ...,
    PRIMARY KEY(`sku`),
    FOREIGN KEY(`sku`) REFERENCES `product`(`sku`))
...
```

Multiple table inheritance is more space-efficient than *single table inheritance* (page 340) and somewhat more flexible than *concrete table inheritance* (page 340). However, this model does require an expensive `JOIN` operation to obtain all relevant attributes relevant to a product.

Entity Attribute Values

The final substantive pattern from relational modeling is the entity-attribute-value schema where you would create a meta-model for product data. In this approach, you maintain a table with three columns, e.g. `entity_id`, `attribute_id`, `value`, and these triples describe each product.

Consider the description of an audio recording. You may have a series of rows representing the following relationships:

Entity	Attribute	Value
sku_00e8da9b	type	Audio Album
sku_00e8da9b	title	A Love Supreme
sku_00e8da9b	...	...
sku_00e8da9b	artist	John Coltrane
sku_00e8da9b	genre	Jazz
sku_00e8da9b	genre	General
...	...	...

This schema is totally flexible:

- any entity can have any set of any attributes.
- New product categories do not require *any* changes to the data model in the database.

The downside for these models, is that all nontrivial queries require large numbers of `JOIN` operations that results in large performance penalties.

Avoid Modeling Product Data

Additionally some e-commerce solutions with relational database systems avoid choosing one of the the data models above, and serialize all of this data into a `BLOB` column. While simple, the details become difficult to access for search and sort.

Non-Relational Data Model

Because MongoDB is a non-relational database, the data model for your product catalog can benefit from this additional flexibility. The best models use a single MongoDB collection to store all the product data, which is similar to the

single table inheritance (page 340) relational model. MongoDB's dynamic schema means that each *document* need not conform to the same schema. As a result, the document for each product only needs to contain attributes relevant to that product.

Schema

At the beginning of the document, the schema must contain general product information, to facilitate searches of the entire catalog. Then, a `details` sub-document that contains fields that vary between product types. Consider the following example document for an album product.

```
{
  sku: "00e8da9b",
  type: "Audio Album",
  title: "A Love Supreme",
  description: "by John Coltrane",
  asin: "B0000A118M",

  shipping: {
    weight: 6,
    dimensions: {
      width: 10,
      height: 10,
      depth: 1
    },
  },

  pricing: {
    list: 1200,
    retail: 1100,
    savings: 100,
    pct_savings: 8
  },

  details: {
    title: "A Love Supreme [Original Recording Reissued]",
    artist: "John Coltrane",
    genre: [ "Jazz", "General" ],
        ...
    tracks: [
      "A Love Supreme Part I: Acknowledgement",
      "A Love Supreme Part II - Resolution",
      "A Love Supreme, Part III: Pursuance",
      "A Love Supreme, Part IV-Psalm"
    ],
  },
}
```

A movie item would have the same fields for general product information, shipping, and pricing, but have different details sub-document. Consider the following:

```
{
  sku: "00e8da9d",
  type: "Film",
  ...,
  asin: "B000P0J0AQ",

  shipping: { ... },
```

```
  pricing: { ... },

  details: {
    title: "The Matrix",
    director: [ "Andy Wachowski", "Larry Wachowski" ],
    writer: [ "Andy Wachowski", "Larry Wachowski" ],
    ...,
    aspect_ratio: "1.66:1"
  },
}
```

Note: In MongoDB, you can have fields that hold multiple values (i.e. arrays) without any restrictions on the number of fields or values (as with `genre_0` and `genre_1`) and also without the need for a JOIN operation.

27.1.2 Operations

For most deployments the primary use of the product catalog is to perform search operations. This section provides an overview of various types of queries that may be useful for supporting an e-commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Find Albums by Genre and Sort by Year Produced

Querying

This query returns the documents for the products of a specific genre, sorted in reverse chronological order:

```
query = db.products.find({'type':'Audio Album',
                          'details.genre': 'jazz'})
query = query.sort([('details.issue_date', -1)])
```

Indexing

To support this query, create a compound index on all the properties used in the filter and in the sort:

```
db.products.ensure_index([
    ('type', 1),
    ('details.genre', 1),
    ('details.issue_date', -1)])
```

Note: The final component of the index is the sort field. This allows MongoDB to traverse the index in the sorted order to preclude a slow in-memory sort.

Find Products Sorted by Percentage Discount Descending

While most searches will be for a particular type of product (e.g album, movie, etc.,) in some situations you may want to return all products in a certain price range, or discount percentage.

Querying

To return this data use the pricing information that exists in all products to find the products with the highest percentage discount:

```
query = db.products.find( { 'pricing.pct_savings': {'$gt': 25 })
query = query.sort([('pricing.pct_savings', -1)])
```

Indexing

To support this type of query, you will want to create an index on the `pricing.pct_savings` field:

```
db.products.ensure_index('pricing.pct_savings')
```

Since MongoDB can read indexes in ascending or descending order, the order of the index does not matter.

Note: If you want to preform range queries (e.g. "return all products over $25") and then sort by another property like `pricing.retail`, MongoDB cannot use the index as effectively in this situation.

The field that you want to select a range, or perform sort operations, must be the *last* field in a compound index in order to avoid scanning an entire collection. Using different properties within a single combined range query and sort operation requires some scanning which will limit the speed of your query.

Find Movies Based on Staring Actor

Querying

Use the following query to select documents within the details of a specified product type (i.e. `Film`) of product (a movie) to find products that contain a certain value (i.e. a specific actor in the `details.actor` field,) with the results sorted by date descending:

```
query = db.products.find({'type': 'Film',
                          'details.actor': 'Keanu Reeves'})
query = query.sort([('details.issue_date', -1)])
```

Indexing

To support this query, you may want to create the following index.

```
db.products.ensure_index([
    ('type', 1),
    ('details.actor', 1),
    ('details.issue_date', -1)])
```

This index begins with the `type` field and then narrows by the other search field, where the final component of the index is the sort field to maximize index efficiency.

Find Movies with a Particular Word in the Title

Regardless of database engine, in order to retrieve this information the system will need to scan some number of documents or records to satisfy this query.

Querying

MongoDB supports regular expressions within queries. In Python, you can use the "re" module to construct the query:

```
import re
re_hacker = re.compile(r'.*hacker.*', re.IGNORECASE)

query = db.products.find({'type': 'Film', 'title': re_hacker})
query = query.sort([('details.issue_date', -1)])
```

MongoDB provides a special syntax for regular expression queries without the need for the re module. Consider the following alternative which is equivalent to the above example:

```
query = db.products.find({
    'type': 'Film',
    'title': {'$regex': '.*hacker.*', '$options':'i'}})
query = query.sort([('details.issue_date', -1)])
```

The $options operator specifies a case insensitive match.

Indexing

The indexing strategy for these kinds of queries is different from previous attempts. Here, create an index on { type: 1, details.issue_date: -1, title: 1 } using the following command at the Python/PyMongo console:

```
db.products.ensure_index([
    ('type', 1),
    ('details.issue_date', -1),
    ('title', 1)])
```

This index makes it possible to avoid scanning whole documents by using the index for scanning the title rather than forcing MongoDB to scan whole documents for the title field. Additionally, to support the sort on the details.issue_date field, by placing this field *before* the title field, ensures that the result set is already ordered before MongoDB filters title field.

27.1.3 Scaling

Sharding

Database performance for these kinds of deployments are dependent on indexes. You may use *sharding* to enhance performance by allowing MongoDB to keep larger portions of those indexes in RAM. In sharded configurations, select a *shard key* that allows mongos (page 676) to route queries directly to a single shard or small group of shards.

Since most of the queries in this system include the type field, include this in the shard key. Beyond this, the remainder of the shard key is difficult to predict without information about your database's actual activity and distribution. Consider that:

- details.issue_date would be a poor addition to the shard key because, although it appears in a number of queries, no query was were *selective* by this field.
- you should include one or more fields in the detail document that you query frequently, and a field that has quasi-random features, to prevent large unsplitable chunks.

In the following example, assume that the `details.genre` field is the second-most queried field after `type`. Enable sharding using the following `shardCollection` operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'product', {
...     key : { 'type': 1, 'details.genre' : 1, 'sku':1 } })
{ "collectionsharded" : "details.genre", "ok" : 1 }
```

Note: Even if you choose a "poor" shard key that requires `mongos` (page 676) to broadcast all to all shards, you will still see some benefits from sharding, because:

1. Sharding makes a larger amount of memory available to store indexes, and
2. MongoDB will parallelize queries across shards, reducing latency.

Read Preference

While *sharding* is the best way to scale operations, some data sets make it impossible to partition data so that `mongos` (page 676) can route queries to specific shards. In these situations `mongos` (page 676) sends the query to all shards and then combines the results before returning to the client.

In these situations, you can add additional read performance by allowing `mongos` (page 676) to read from the *secondary* instances in a *replica set* by configuring *read preference* in your client. Read preference is configurable on a per-connection or per-operation basis. In PyMongo, set the `read_preference` argument.

The `SECONDARY` property in the following example, permits reads from a *secondary* (as well as a primary) for the entire connection .

```
conn = pymongo.Connection(read_preference=pymongo.SECONDARY)
```

Conversely, the `SECONDARY_ONLY` read preference means that the client will only send read operation only to the secondary member

```
conn = pymongo.Connection(read_preference=pymongo.SECONDARY_ONLY)
```

You can also specify `read_preference` for specific queries, as follows:

```
results = db.product.find(..., read_preference=pymongo.SECONDARY)
```

or

```
results = db.product.find(..., read_preference=pymongo.SECONDARY_ONLY)
```

See Also:

"*Replica Set Read Preference* (page 55)."

27.2 Inventory Management

27.2.1 Overview

This case study provides an overview of practices and patterns for designing and developing the inventory management portions of an E-commerce application.

See Also:

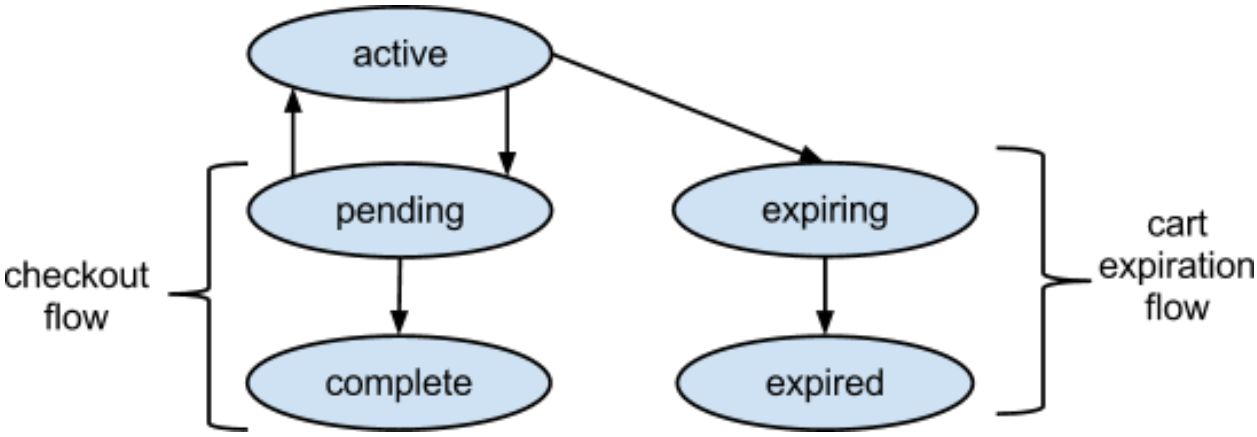
"*Product Catalog* (page 339)."

Problem

Customers in e-commerce stores regularly add and remove items from their "shopping cart," change quantities multiple times, abandon the cart at any point, and sometimes have problems during and after checkout that require a hold or canceled order. These activities make it difficult to maintain inventory systems and counts and ensure that customers cannot "buy" items that are unavailable while they shop in your store.

Solution

This solution keeps the traditional metaphor of the shopping cart, but the shopping cart will *age*. After a shopping cart has been inactive for a certain period of time, all items in the cart re-enter the available inventory and the cart is empty. The state transition diagram for a shopping cart is below:



Schema

Inventory collections must maintain counts of the current available inventory of each stock-keeping unit (SKU; or item) as well as a list of items in carts that may return to the available inventory if they are in a shopping cart that times out. In the following example, the `_id` field stores the SKU:

```
{
    _id: '00e8da9b',
    qty: 16,
    carted: [
        { qty: 1, cart_id: 42,
          timestamp: ISODate("2012-03-09T20:55:36Z"), },
        { qty: 2, cart_id: 43,
          timestamp: ISODate("2012-03-09T21:55:36Z"), },
    ]
}
```

Note: These examples use a simplified schema. In a production implementation, you may choose to merge this schema with the product catalog schema described in the "*Product Catalog* (page 339)" document.

The SKU above has 16 items in stock, 1 item a cart, and 2 items in a second cart. This leaves a total of 19 unsold items of merchandise.

To model the shopping cart objects, you need to maintain `sku`, `quantity`, fields embedded in a shopping cart *document*:

```
{
    _id: 42,
    last_modified: ISODate("2012-03-09T20:55:36Z"),
    status: 'active',
    items: [
        { sku: '00e8da9b', qty: 1, item_details: {...} },
        { sku: '0ab42f88', qty: 4, item_details: {...} }
    ]
}
```

Note: The `item_details` field in each line item allows your application to display the cart contents to the user without requiring a second query to fetch details from the catalog collection.

27.2.2 Operations

This section introduces operations that you may use to support an e-commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Add an Item to a Shopping Cart

Moving an item from the available inventory to a cart is a fundamental requirement for a shopping cart system. The most important requirement is to ensure that your application will never move an unavailable item from the inventory to the cart.

Ensure that inventory is only updated if there is sufficient inventory to satisfy the request with the following `add_item_to_cart` function operation.

```
def add_item_to_cart(cart_id, sku, qty, details):
    now = datetime.utcnow()

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active' },
        { '$set': { 'last_modified': now },
          '$push': {
              'items': {'sku': sku, 'qty':qty, 'details': details } } },
        safe=True)
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.inventory.update(
        {'_id':sku, 'qty': {'$gte': qty}},
        {'$inc': {'qty': -qty},
         '$push': {
             'carted': { 'qty': qty, 'cart_id':cart_id,
                         'timestamp': now } } },
        safe=True)
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id },
```

```
            { '$pull': { 'items': {'sku': sku } } })
        raise InadequateInventory()
```

The system does not trust that the available inventory can satisfy a request

First this operation checks to make sure that that the cart is "active" before adding a item. Then, it verifies that the available inventory to satisfy the request before decrementing inventory.

If there is not adequate inventory, the system removes the cart update: by specifying `safe=True` and checking the result allows the application to report an error if the cart is inactive or available quantity is insufficient to satisfy the request.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Modifying the Quantity in the Cart

The following process underlies adjusting the quantity of items in a users cart. The application must ensure that when a user increases the quantity of an item, in addition to updating the `carted` entry for the user's cart, that the inventory exists to cover the modification.

```
def update_quantity(cart_id, sku, old_qty, new_qty):
    now = datetime.utcnow()
    delta_qty = new_qty - old_qty

    # Make sure the cart is still active and add the line item
    result = db.cart.update(
        {'_id': cart_id, 'status': 'active', 'items.sku': sku },
        {'$set': {
             'last_modified': now,
             'items.$.qty': new_qty },
        },
        safe=True)
    if not result['updatedExisting']:
        raise CartInactive()

    # Update the inventory
    result = db.inventory.update(
        {'_id':sku,
         'carted.cart_id': cart_id,
         'qty': {'$gte': delta_qty} },
        {'$inc': {'qty': -delta_qty },
         '$set': { 'carted.$.qty': new_qty, 'timestamp': now } },
        safe=True)
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id, 'items.sku': sku },
            {'$set': { 'items.$.qty': old_qty } })
        raise InadequateInventory()
```

Note: That the positional operator `$` updates the particular `carted` entry and item that matched the query.

This allows the application to update the inventory and keep track of the data needed to "rollback" the cart in a single atomic operation. The code also ensures that the cart is active.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Checking Out

The checkout operation must: validate the method of payment and remove the `carted` items after the transaction succeeds. Consider the following procedure:

```
def checkout(cart_id):
    now = datetime.utcnow()

    # Make sure the cart is still active and set to 'pending'. Also
    #     fetch the cart details so we can calculate the checkout price
    cart = db.cart.find_and_modify(
        {'_id': cart_id, 'status': 'active' },
        update={'$set': { 'status': 'pending','last_modified': now } } )
    if cart is None:
        raise CartInactive()

    # Validate payment details; collect payment
    try:
        collect_payment(cart)
        db.cart.update(
            {'_id': cart_id },
            {'$set': { 'status': 'complete' } } )
        db.inventory.update(
            {'carted.cart_id': cart_id},
            {'$pull': {'cart_id': cart_id} },
            multi=True)
    except:
        db.cart.update(
            {'_id': cart_id },
            {'$set': { 'status': 'active' } } )
        raise
```

Begin by "locking" the cart by setting its status to "pending" Then the system will verify that the cart is still active and collect payment data. Then, the `findAndModify` *command* makes it possible to update the cart atomically and return its details to capture payment information. Then:

- If the payment is successful, then the application will remove the `carted` items from the inventory documents and set the cart to `complete`.
- If payment is unsuccessful, the application will unlock the cart by setting its status to `active` and report a payment error.

Note: This operation requires no *indexes* beyond the default index on the `_id` field.

Returning Inventory from Timed-Out Carts

Process

Periodically, your application must "expire" inactive carts and return their items to available inventory. In the example that follows the variable `timeout` controls the length of time before a cart expires:

```
def expire_carts(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Lock and find all the expiring carts
    db.cart.update(
        {'status': 'active', 'last_modified': { '$lt': threshold } },
        {'$set': { 'status': 'expiring' } },
        multi=True )

    # Actually expire each cart
    for cart in db.cart.find({'status': 'expiring'}):

        # Return all line items to inventory
        for item in cart['items']:
            db.inventory.update(
                { '_id': item['sku'],
                  'carted.cart_id': cart['id'],
                  'carted.qty': item['qty']
                },
                {'$inc': { 'qty': item['qty'] },
                 '$pull': { 'carted': { 'cart_id': cart['id'] } } })

        db.cart.update(
            {'_id': cart['id'] },
            {'$set': { status': 'expired' })
```

This procedure:

1. finds all carts that are older than the `threshold` and are due for expiration.
2. for each "expiring" cart, return all items to the available inventory.
3. once the items return to the available inventory, set the `status` field to `expired`.

Indexing

To support returning inventory from timed-out cart, create an index to support queries on their `status` and `last_modified` fields. Use the following operations in the Python/PyMongo shell:

```
db.cart.ensure_index([('status', 1), ('last_modified', 1)])
```

Error Handling

The above operations do not account for one possible failure situation: if an exception occurs after updating the shopping cart but before updating the inventory collection. This would result in a shopping cart that may be absent or expired but items have not returned to available inventory.

To account for this case, your application will need a periodic cleanup operation that finds inventory items that have `carted` items and check that to ensure that they exist in a user's cart, and return them to available inventory if they do not.

```
def cleanup_inventory(timeout):
    now = datetime.utcnow()
    threshold = now - timedelta(seconds=timeout)

    # Find all the expiring carted items
```

```
for item in db.inventory.find(
    {'carted.timestamp': {'$lt': threshold }}):

    # Find all the carted items that matched
    carted = dict(
              (carted_item['cart_id'], carted_item)
              for carted_item in item['carted']
              if carted_item['timestamp'] < threshold)

    # First Pass: Find any carts that are active and refresh the carted items
    for cart in db.cart.find(
        { '_id': {'$in': carted.keys() },
        'status':'active'}):
        cart = carted[cart['_id']]

        db.inventory.update(
            { '_id': item['_id'],
              'carted.cart_id': cart['_id'] },
            { '$set': {'carted.$.timestamp': now } })
        del carted[cart['_id']]

    # Second Pass: All the carted items left in the dict need to now be
    #    returned to inventory
    for cart_id, carted_item in carted.items():
        db.inventory.update(
            { '_id': item['_id'],
              'carted.cart_id': cart_id,
              'carted.qty': carted_item['qty'] },
            { '$inc': { 'qty': carted_item['qty'] },
              '$pull': { 'carted': { 'cart_id': cart_id } } })
```

To summarize: This operation finds all "carted" items that have time stamps older than the threshold. Then, the process makes two passes over these items:

1. Of the items with time stamps older than the threshold, if the cart is still active, it resets the time stamp to maintain the carts.
2. Of the stale items that remain in inactive carts, the operation returns these items to the inventory.

Note: The function above is safe for use because it checks to ensure that the cart has expired before returning items from the cart to inventory. However, it could be long-running and slow other updates and queries.

Use judiciously.

27.2.3 Sharding

If you need to *shard* the data for this system, the `_id` field is an ideal *shard key* for both carts and products because most update operations use the `_id` field. This allows `mongos` (page 676) to route all updates that select on `_id` to a single `mongod` process.

There are two drawbacks for using `_id` as a shard key:

- If the cart collection's `_id` is an incrementing value, all new carts end up on a single shard.

  You can mitigate this effect by choosing a random value upon the creation of a cart, such as a hash (i.e. MD5 or SHA-1) of an ObjectID, as the `_id`. The process for this operation would resemble the following:

```
import hashlib
import bson

cart_id = bson.ObjectId()
cart_id_hash = hashlib.md5(str(cart_id)).hexdigest()

cart = { "_id": cart_id, "cart_hash": cart_id_hash }
db.cart.insert(cart)
```

- Cart expiration and inventory adjustment requires update operations and queries to broadcast to all shards when using `_id` as a shard key.

  This may be less relevant as the expiration functions run relatively infrequently and you can queue them or artificially slow them down (as with judicious use of `sleep()`) to minimize server load.

Use the following commands in the Python/PyMongo console to shard the cart and inventory collections:

```
>>> db.command('shardCollection', 'inventory'
...             'key': { '_id': 1 } )
{ "collectionsharded" : "inventory", "ok" : 1 }
>>> db.command('shardCollection', 'cart')
...             'key': { '_id': 1 } )
{ "collectionsharded" : "cart", "ok" : 1 }
```

27.3 Category Hierarchy

27.3.1 Overview

This document provides the basic design for modeling a product hierarchy stored in MongoDB as well as a collection of common operations for interacting with this data that will help you begin to write an E-commerce product category hierarchy.

See Also:

"*Product Catalog* (page 339)"

Solution

To model a product category hierarchy, this solution keeps each category in its own document that also has a list of its ancestors or "parents." This document uses music genres as the basis of its examples:

Because these kinds of categories change infrequently, this model focuses on the operations needed to keep the hierarchy up-to-date rather than the performance profile of update operations.

Schema

This schema has the following properties:

- A single document represents each category in the hierarchy.
- An `ObjectId` identifies each category document for internal cross-referencing.
- Each category document has a human-readable name and a URL compatible `slug` field.
- The schema stores a list of ancestors for each category to facilitate displaying a query and its ancestors using only a single query.

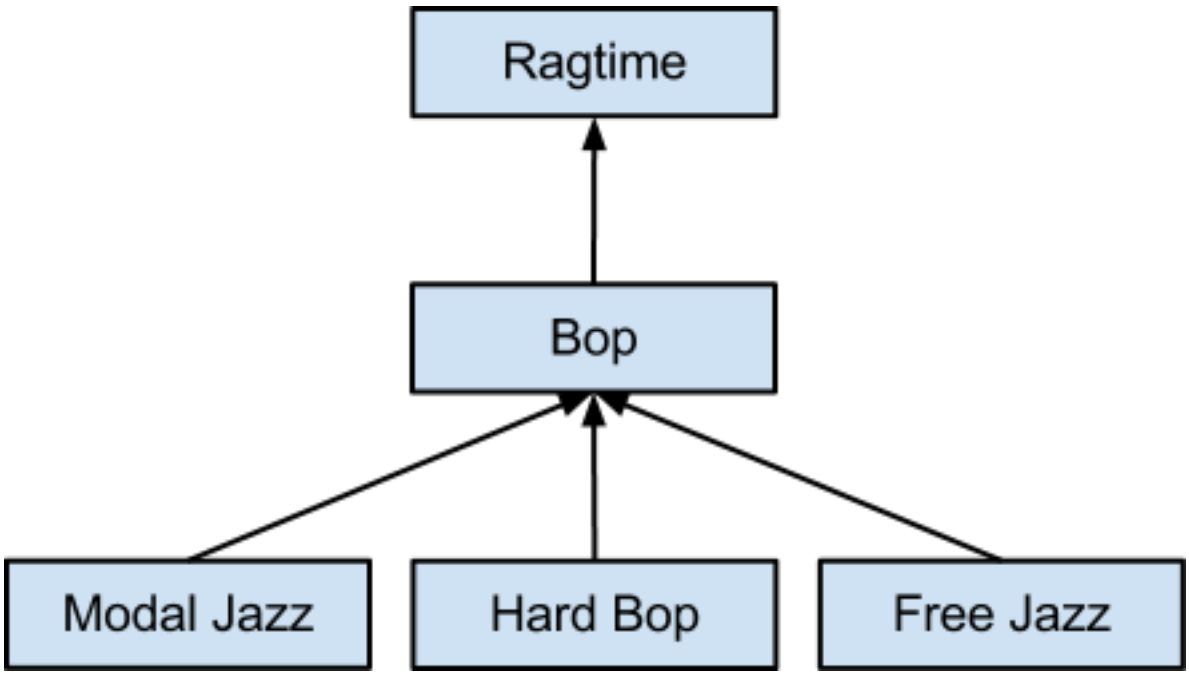


Figure 27.1: Initial category hierarchy

Consider the following prototype:

```
{ "_id" : ObjectId("4f5ec858eb03303a11000002"),
  "name" : "Modal Jazz",
  "parent" : ObjectId("4f5ec858eb03303a11000001"),
  "slug" : "modal-jazz",
  "ancestors" : [
         { "_id" : ObjectId("4f5ec858eb03303a11000001"),
        "slug" : "bop",
        "name" : "Bop" },
         { "_id" : ObjectId("4f5ec858eb03303a11000000"),
           "slug" : "ragtime",
           "name" : "Ragtime" } ]
}
```

27.3.2 Operations

This section outlines the category hierarchy manipulations that you may need in an E-Commerce site. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Read and Display a Category

Querying

Use the following option to read and display a category hierarchy. This query will use the `slug` field to return the category information and a "bread crumb" trail from the current category to the top level category.

```
category = db.categories.find(
    {'slug':slug},
    {'_id':0, 'name':1, 'ancestors.slug':1, 'ancestors.name':1 })
```

Indexing

Create a unique index on the `slug` field with the following operation on the Python/PyMongo console:

```
>>> db.categories.ensure_index('slug', unique=True)
```

Add a Category to the Hierarchy

To add a category you must first determine its ancestors. Take adding a new category "Swing" as a child of "Ragtime", as below:

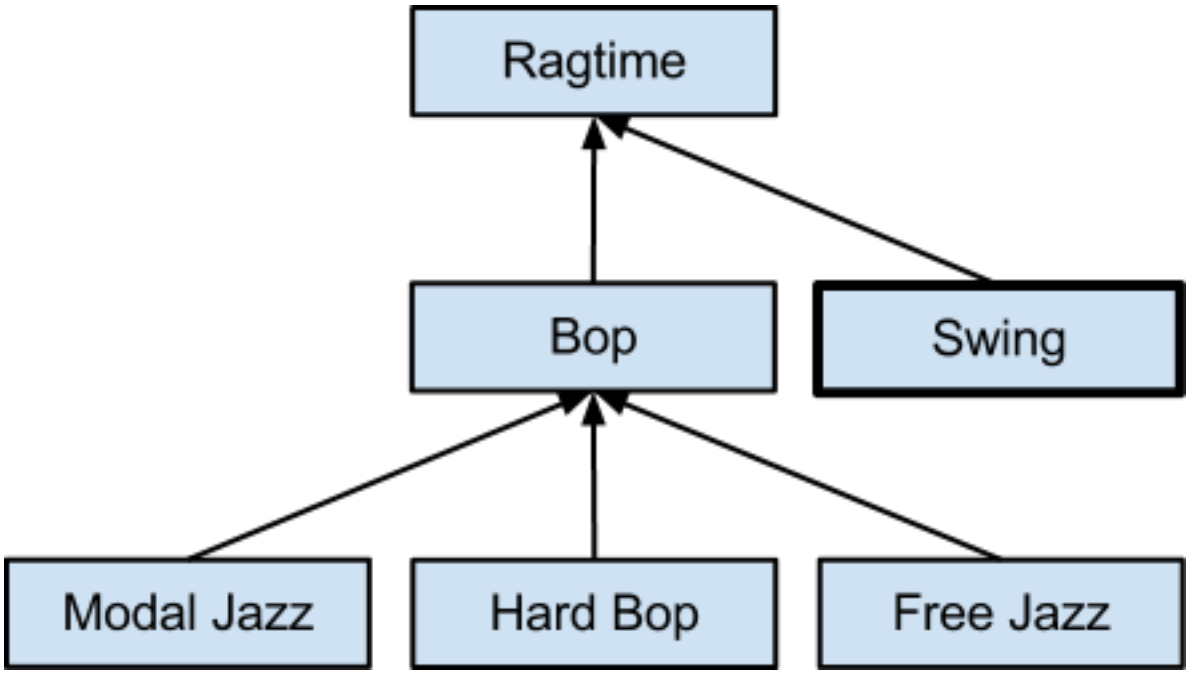


Figure 27.2: Adding a category

The insert operation would be trivial except for the ancestors. To define this array, consider the following helper function:

```
def build_ancestors(_id, parent_id):
    parent = db.categories.find_one(
        {'_id': parent_id},
        {'name': 1, 'slug': 1, 'ancestors':1})
    parent_ancestors = parent.pop('ancestors')
    ancestors = [ parent ] + parent_ancestors
    db.categories.update(
        {'_id': _id},
        {'$set': { 'ancestors': ancestors } })
```

You only need to travel "up" one level in the hierarchy to get the ancestor list for "Ragtime" that you can use to build the ancestor list for "Swing." Then create ate document with the following set of operations:

```
doc = dict(name='Swing', slug='swing', parent=ragtime_id)
swing_id = db.categories.insert(doc)
build_ancestors(swing_id, ragtime_id)
```

Note: Since these queries and updates all selected based on `_id`, you only need the default MongoDB-supplied index on `_id` to support this operation efficiently.

Change the Ancestry of a Category

This section address the process for reorganizing the hierarchy by moving "bop" under "swing" as follows:

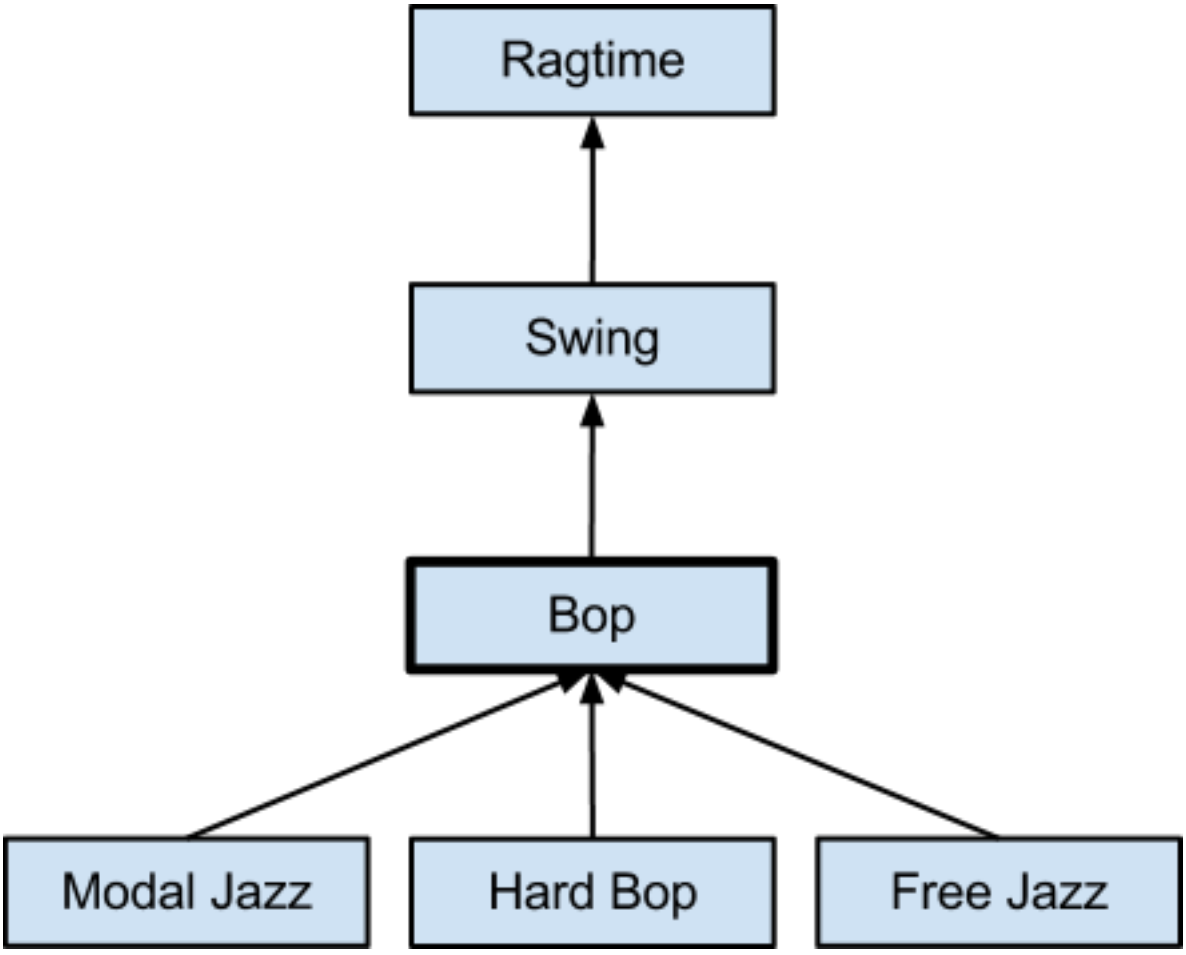


Figure 27.3: Change the parent of a category

Procedure

Update the `bop` document to reflect the change in ancestry with the following operation:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'parent': swing_id } } )
```

The following helper function, rebuilds the ancestor fields to ensure correctness. [1]

```
def build_ancestors_full(_id, parent_id):
    ancestors = []
    while parent_id is not None:
        parent = db.categories.find_one(
            {'_id': parent_id},
            {'parent': 1, 'name': 1, 'slug': 1, 'ancestors':1})
        parent_id = parent.pop('parent')
        ancestors.append(parent)
    db.categories.update(
        {'_id': _id},
        {'$set': { 'ancestors': ancestors } })
```

You can use the following loop to reconstruct all the descendants of the "bop" category:

```
for cat in db.categories.find(
    {'ancestors._id': bop_id},
    {'parent_id': 1}):
    build_ancestors_full(cat['_id'], cat['parent_id'])
```

Indexing

Create an index on the `ancestors._id` field to support the update operation.

```
db.categories.ensure_index('ancestors._id')
```

Rename a Category

To a rename a category you need to both update the category itself and also update all the descendants. Consider renaming "Bop" to "BeBop" as in the following figure:

First, you need to update the category name with the following operation:

```
db.categories.update(
    {'_id':bop_id}, {'$set': { 'name': 'BeBop' } } )
```

Next, you need to update each descendant's ancestors list:

```
db.categories.update(
    {'ancestors._id': bop_id},
    {'$set': { 'ancestors.$.name': 'BeBop' } },
    multi=True)
```

This operation uses:

- the positional operation `$` to match the exact "ancestor" entry that matches the query, and
- the `multi` option to update all documents that match this query.

Note: In this case, the index you have already defined on `ancestors._id` is sufficient to ensure good performance.

---

[1] Your application cannot guarantee that the ancestor list of a parent category is correct, because MongoDB may process the categories out-of-order.

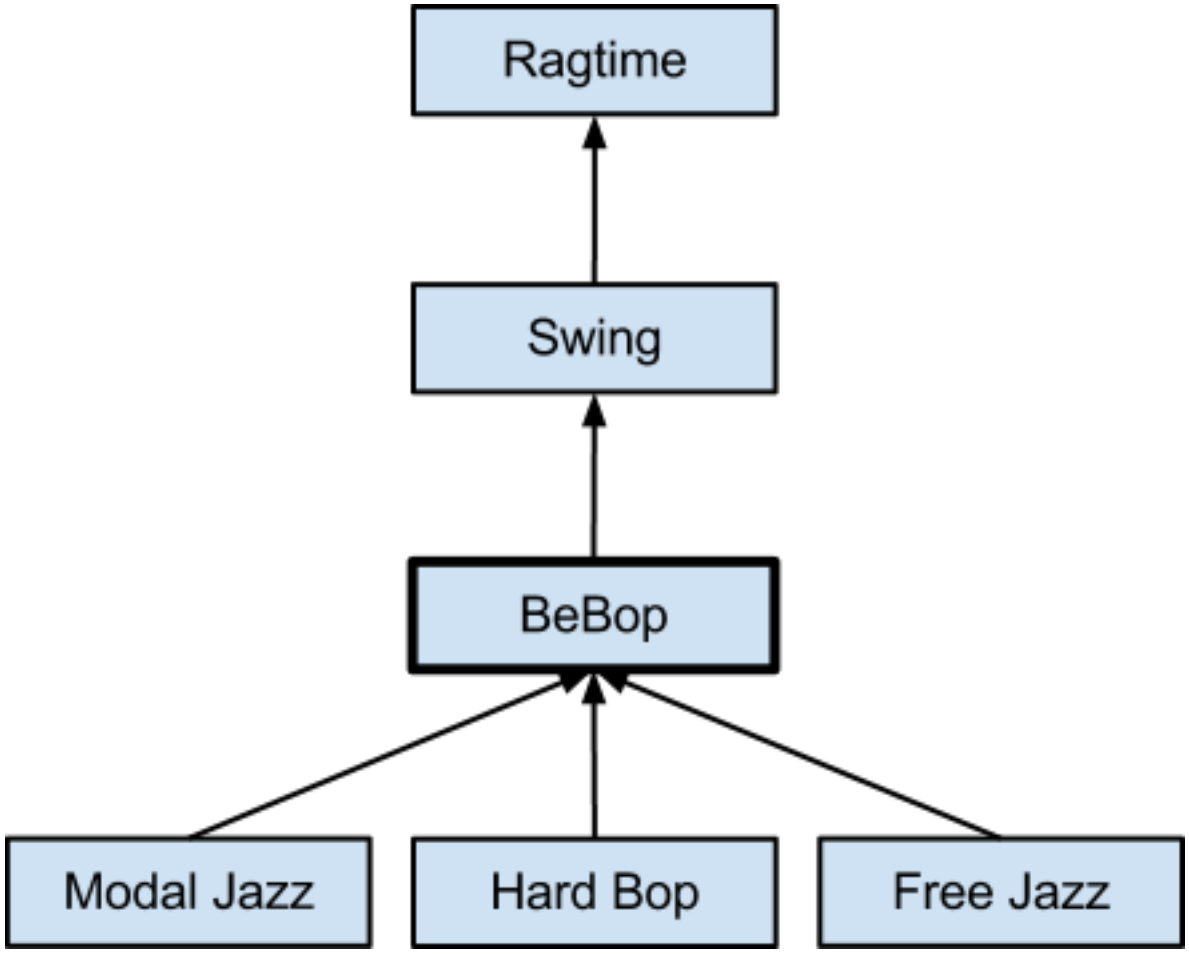


Figure 27.4: Rename a category

27.3.3 Sharding

For most deployments, *sharding* this collection has limited value because the collection will be very small. If you do need to shard, because most updates query the `_id` field, this field is a suitable *shard key*. Shard the collection with the following operation in the Python/PyMongo console.

```
>>> db.command('shardCollection', 'categories', {
...     'key': {'_id': 1} })
{ "collectionsharded" : "categories", "ok" : 1 }
```

CHAPTER

TWENTYEIGHT

CONTENT MANAGEMENT SYSTEMS

The content management use cases introduce fundamental MongoDB practices and approaches, using familiar problems and simple examples. The "*Metadata and Asset Management* (page 361)" document introduces a model that you may use when designing a web site content management system, while "*Storing Comments* (page 368)" introduces the method for modeling user comments on content, like blog posts, and media, in MongoDB.

28.1 Metadata and Asset Management

28.1.1 Overview

This document describes the design and pattern of a content management system using MongoDB modeled on the popular Drupal CMS.

Problem

You are designing a content management system (CMS) and you want to use MongoDB to store the content of your sites.

Solution

To build this system you will use MongoDB's flexible schema to store all content "nodes" in a single collection regardless of type. This guide will provide prototype schema and describe common operations for the following primary node types:

Basic Page Basic pages are useful for displaying infrequently-changing text such as an 'about' page. With a basic page, the salient information is the title and the content.

Blog entry Blog entries record a "stream" of posts from users on the CMS and store title, author, content, and date as relevant information.

Photo Photos participate in photo galleries, and store title, description, author, and date along with the actual photo binary data.

This solution does not describe schema or process for storing or using navigational and organizational information.

Schema

Although *documents* in the `nodes` collection contain content of different times, all documents have a similar structure and a set of common fields. Consider the following prototype document for a "basic page" node type:

```
{
    _id: ObjectId(...),
    nonce: ObjectId(...),
    metadata: {
        type: 'basic-page'
        section: 'my-photos',
        slug: 'about',
        title: 'About Us',
        created: ISODate(...),
        author: { _id: ObjectId(...), name: 'Rick' },
        tags: [ ... ],
        detail: { text: '# About Us\n...' }
    }
}
```

Most fields are descriptively titled. The `section` field identifies groupings of items, as in a photo gallery, or a particular blog . The `slug` field holds a URL-friendly unique representation of the node, usually that is unique within its section for generating URLs.

All documents also have a `detail` field that varies with the document type. For the basic page above, the detail field might hold the text of the page. For a blog entry, the `detail` field might hold a sub-document. Consider the following prototype:

```
{
    ...
    metadata: {
        ...
        type: 'blog-entry',
        section: 'my-blog',
        slug: '2012-03-noticed-the-news',
        ...
        detail: {
            publish_on: ISODate(...),
            text: 'I noticed the news from Washington today...'
        }
    }
}
```

Photos require a different approach. Because photos can be potentially larger than these documents, it's important to separate the binary photo storage from the nodes metadata.

GridFS provides the ability to store larger files in MongoDB. GridFS stores data in two collections, in this case, `cms.assets.files`, which stores metadata, and `cms.assets.chunks` which stores the data itself. Consider the following prototype document from the `cms.assets.files` collection:

```
{
    _id: ObjectId(...),
    length: 123...,
    chunkSize: 262144,
    uploadDate: ISODate(...),
    contentType: 'image/jpeg',
    md5: 'ba49a...',
    metadata: {
        nonce: ObjectId(...),
        slug: '2012-03-invisible-bicycle',
        type: 'photo',
        section: 'my-album',
        title: 'Kitteh',
        created: ISODate(...),
```

```
        author: { _id: ObjectId(...), name: 'Jared' },
        tags: [ ... ],
        detail: {
            filename: 'kitteh_invisible_bike.jpg',
            resolution: [ 1600, 1600 ], ... }
    }
}
```

Note: This document embeds the basic node document fields, which allows you to use the same code to manipulate nodes, regardless of type.

28.1.2 Operations

This section outlines a number of common operations for building and interacting with the metadata and asset layer of the cms for all node types. All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Create and Edit Content Nodes

Procedure

The most common operations inside of a CMS center on creating and editing content. Consider the following `insert()` operation:

```
db.cms.nodes.insert({
    'nonce': ObjectId(),
    'metadata': {
        'section': 'myblog',
        'slug': '2012-03-noticed-the-news',
        'type': 'blog-entry',
        'title': 'Noticed in the News',
        'created': datetime.utcnow(),
        'author': { 'id': user_id, 'name': 'Rick' },
        'tags': [ 'news', 'musings' ],
        'detail': {
            'publish_on': datetime.utcnow(),
            'text': 'I noticed the news from Washington today...' }
        }
     })
```

Once inserted, your application must have some way of preventing multiple concurrent updates. The schema uses the special `nonce` field to help detect concurrent edits. By using the `nonce` field in the query portion of the `update` operation, the application will generate an error if there is an editing collision. Consider the following `update`

```
def update_text(section, slug, nonce, text):
    result = db.cms.nodes.update(
        { 'metadata.section': section,
          'metadata.slug': slug,
          'nonce': nonce },
        { '$set':{'metadata.detail.text': text, 'nonce': ObjectId() } },
        safe=True)
    if not result['updatedExisting']:
        raise ConflictError()
```

You may also want to perform metadata edits to the item such as adding tags:

```
db.cms.nodes.update(
    { 'metadata.section': section, 'metadata.slug': slug },
    { '$addToSet': { 'tags': { '$each': [ 'interesting', 'funny' ] } } })
```

In this example the `$addToSet` operator will only add values to the `tags` field if they do not already exist in the `tags` array, there's no need to supply or update the nonce.

Index Support

To support updates and queries on the `metadata.section`, and `metadata.slug`, fields *and* to ensure that two editors don't create two documents with the same section name or slug. Use the following operation at the Python/PyMongo console:

```
>>> db.cms.nodes.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1)], unique=True)
```

The `unique=True` option prevents to documents from colliding. If you want an index to support queries on the above fields and the `nonce` field create the following index:

```
>>> db.cms.nodes.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1), ('nonce', 1) ])
```

However, in most cases, the first index will be sufficient to support these operations.

Upload a Photo

Procedure

To update a photo object, use the following operation, which builds upon the basic update procedure:

```
def upload_new_photo(
    input_file, section, slug, title, author, tags, details):
    fs = GridFS(db, 'cms.assets')
    with fs.new_file(
        content_type='image/jpeg',
        metadata=dict(
            type='photo',
            locked=datetime.utcnow(),
            section=section,
            slug=slug,
            title=title,
            created=datetime.utcnow(),
            author=author,
            tags=tags,
            detail=detail)) as upload_file:
        while True:
            chunk = input_file.read(upload_file.chunk_size)
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update(
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

Because uploading the photo spans multiple documents and is a non-atomic operation, you must "lock" the file during upload by writing `datetime.utcnow()` in the record. This helps when there are multiple concurrent editors and lets the application detect stalled file uploads. This operation assumes that, for photo upload, the last update will succeed:

```
def update_photo_content(input_file, section, slug):
    fs = GridFS(db, 'cms.assets')

    # Delete the old version if it's unlocked or was locked more than 5
    #    minutes ago
    file_obj = db.cms.assets.find_one(
        { 'metadata.section': section,
          'metadata.slug': slug,
          'metadata.locked': None })
    if file_obj is None:
        threshold = datetime.utcnow() - timedelta(seconds=300)
        file_obj = db.cms.assets.find_one(
            { 'metadata.section': section,
              'metadata.slug': slug,
              'metadata.locked': { '$lt': threshold } })
    if file_obj is None: raise FileDoesNotExist()
    fs.delete(file_obj['_id'])

    # update content, keep metadata unchanged
    file_obj['locked'] = datetime.utcnow()
    with fs.new_file(**file_obj):
        while True:
            chunk = input_file.read(upload_file.chunk_size)
            if not chunk: break
            upload_file.write(chunk)
    # unlock the file
    db.assets.files.update(
        {'_id': upload_file._id},
        {'$set': { 'locked': None } } )
```

As with the basic operations, you can use a much more simple operation to edit the tags:

```
db.cms.assets.files.update(
    { 'metadata.section': section, 'metadata.slug': slug },
    { '$addToSet': { 'metadata.tags': { '$each': [ 'interesting', 'funny' ] } } })
```

Index Support

Create a unique index on `{ metadata.section: 1, metadata.slug: 1 }` to support the above operations and prevent users from creating or updating the same file concurrently. Use the following operation in the Python/PyMongo console:

```
>>> db.cms.assets.files.ensure_index([
...     ('metadata.section', 1), ('metadata.slug', 1)], unique=True)
```

Locate and Render a Node

To locate a node based on the value of `metadata.section` and `metadata.slug`, use the following `find_one` operation.

```
node = db.nodes.find_one({'metadata.section': section, 'metadata.slug': slug })
```

Note: The index defined (`section`, `slug`) created to support the update operation, is sufficient to support this operation as well.

Locate and Render a Photo

To locate an image based on the value of `metadata.section` and `metadata.slug`, use the following `find_one` operation.

```
fs = GridFS(db, 'cms.assets')
with fs.get_version({'metadata.section': section, 'metadata.slug': slug }) as img_fpo:
    # do something with the image file
```

Note: The index defined (`section`, `slug`) created to support the update operation, is sufficient to support this operation as well.

Search for Nodes by Tag

Querying

To retrieve a list of nodes based on their tags, use the following query:

```
nodes = db.nodes.find({'metadata.tags': tag })
```

Indexing

Create an index on the `tags` field in the `cms.nodes` collection, to support this query:

```
>>> db.cms.nodes.ensure_index('tags')
```

Search for Images by Tag

Procedure

To retrieve a list of images based on their tags, use the following operation:

```
image_file_objects = db.cms.assets.files.find({'metadata.tags': tag })
fs = GridFS(db, 'cms.assets')
for image_file_object in db.cms.assets.files.find(
    {'metadata.tags': tag }):
    image_file = fs.get(image_file_object['_id'])
    # do something with the image file
```

Indexing

Create an index on the `tags` field in the `cms.assets.files` collection, to support this query:

```
>>> db.cms.assets.files.ensure_index('tags')
```

Generate a Feed of Recently Published Blog Articles

Querying

Use the following operation to generate a list of recent blog posts sorted in descending order by date, for use on the index page of your site, or in an `.rss` or `.atom` feed.

```
articles = db.nodes.find({
    'metadata.section': 'my-blog'
    'metadata.published': { '$lt': datetime.utcnow() } })
articles = articles.sort({'metadata.published': -1})
```

Note: In many cases you will want to limit the number of nodes returned by this query.

Indexing

Create an compound index the the `{ metadata.section: 1, metadata.published: 1 }` fields to support this query and sort operation.

```
>>> db.cms.nodes.ensure_index(
...     [ ('metadata.section', 1), ('metadata.published', -1) ])
```

Note: For all sort or range queries, ensure that field with the sort or range operation is the final field in the index.

28.1.3 Sharding

In a CMS, read performance is more critical than write performance. To achieve the best read performance in a *sharded cluster*, ensure that the `mongos` (page 676) can route queries to specific *shards*.

Also remember that MongoDB can not enforce unique indexes across shards. Using a compound *shard key* that consists of `metadata.section` and `metadata.slug`, will provide the same semantics as describe above.

Warning: Consider the actual use and workload of your cluster before configuring sharding for your cluster.

Use the following operation at the Python/PyMongo shell:

```
>>> db.command('shardCollection', 'cms.nodes', {
...     key : { 'metadata.section': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "cms.nodes", "ok": 1}
>>> db.command('shardCollection', 'cms.assets.files', {
...     key : { 'metadata.section': 1, 'metadata.slug' : 1 } })
{ "collectionsharded": "cms.assets.files", "ok": 1}
```

To shard the `cms.assets.chunks` collection, you must use the `_id` field as the *shard key*. The following operation will shard the collection

```
>>> db.command('shardCollection', 'cms.assets.chunks', {
...     key : { 'files_id': 1 } })
{ "collectionsharded": "cms.assets.chunks", "ok": 1}
```

Sharding on the `files_id` field ensures routable queries because all reads from GridFS must first look up the document in `cms.assets.files` and then look up the chunks separately.

28.2 Storing Comments

This document outlines the basic patterns for storing user-submitted comments in a content management system (CMS.)

28.2.1 Overview

MongoDB provides a number of different approaches for storing data like users-comments on content from a CMS. There is no correct implementation, but there are a number of common approaches and known considerations for each approach. This case study explores the implementation details and trade offs of each option. The three basic patterns are:

1. Store each comment in its own *document*.

   This approach provides the greatest flexibility at the expense of some additional application level complexity.

   These implementations make it possible to display comments in chronological or threaded order, and place no restrictions on the number of comments attached to a specific object.

2. Embed all comments in the "parent" document.

   This approach provides the greatest possible performance for displaying comments at the expense of flexibility: the structure of the comments in the document controls the display format.

   Note: Because of the *limit on document size* (page 679), documents, including the original content and all comments, cannot grow beyond 16 megabytes.

   ---

3. A hybrid design, stores comments separately from the "parent," but aggregates comments into a small number of documents, where each contains many comments.

Also consider that comments can be *threaded*, where comments are always replies to "parent" item or to another comment, which carries certain architectural requirements discussed below.

28.2.2 One Document per Comment

Schema

If you store each comment in its own document, the documents in your `comments` collection, would have the following structure:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    slug: '34db',
    posted: ISODateTime(...),
    author: {
```

```
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This form is only suitable for displaying comments in chronological order. Comments store:

- the `discussion_id` field that references the discussion parent,
- a URL-compatible `slug` identifier,
- a `posted` timestamp,
- an `author` sub-document that contains a reference to a user's profile in the `id` field and their name in the `name` field, and
- the full `text` of the comment.

To support threaded comments, you might use a slightly different structure like the following:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    parent_id: ObjectId(...),
    slug: '34db/8bda'
    full_slug: '2012.02.08.12.21.08:34db/2012.02.09.22.19.16:8bda',
    posted: ISODateTime(...),
    author: {
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This structure:

- adds a `parent_id` field that stores the contents of the `_id` field of the parent comment,
- modifies the `slug` field to hold a path composed of the parent or parent's slug and this comment's unique slug, and
- adds a `full_slug` field that that combines the slugs and time information to make it easier to sort documents in a threaded discussion by date.

Warning: MongoDB can only index *1024 bytes* (page 680). This includes all field data, the field name, and the namespace (i.e. database name and collection name.) This may become an issue when you create an index of the `full_slug` field to support sorting.

Operations

This section contains an overview of common operations for interacting with comments represented using a schema where each comment is its own *document*.

All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose. Issue the following commands at the interactive Python shell to load the required libraries:

```
>>> import bson
>>> import pymongo
```

Post a New Comment

To post a new comment in a chronologically ordered (i.e. without threading) system, use the following `insert()` operation:

```
slug = generate_pseudorandom_slug()
db.comments.insert({
    'discussion_id': discussion_id,
    'slug': slug,
    'posted': datetime.utcnow(),
    'author': author_info,
    'text': comment_text })
```

To insert a comment for a system with threaded comments, you must generate the `slug` path and `full_slug` at insert. See the following operation:

```
posted = datetime.utcnow()

# generate the unique portions of the slug and full_slug
slug_part = generate_pseudorandom_slug()
full_slug_part = posted.strftime('%Y.%m.%d.%H.%M.%S') + ':' + slug_part
# load the parent comment (if any)
if parent_slug:
    parent = db.comments.find_one(
        {'discussion_id': discussion_id, 'slug': parent_slug })
    slug = parent['slug'] + '/' + slug_part
    full_slug = parent['full_slug'] + '/' + full_slug_part
else:
    slug = slug_part
    full_slug = full_slug_part

# actually insert the comment
db.comments.insert({
    'discussion_id': discussion_id,
    'slug': slug,
    'full_slug': full_slug,
    'posted': posted,
    'author': author_info,
    'text': comment_text })
```

View Paginated Comments

To view comments that are not threaded, select all comments participating in a discussion and sort by the `posted` field. For example:

```
cursor = db.comments.find({'discussion_id': discussion_id})
cursor = cursor.sort('posted')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

Because the `full_slug` field contains both hierarchical information (via the path) and chronological information, you can use a simple sort on the `full_slug` field to retrieve a threaded view:

```
cursor = db.comments.find({'discussion_id': discussion_id})
cursor = cursor.sort('full_slug')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

See Also:

`cursor.limit`, `cursor.skip`, and `cursor.sort`

Indexing

To support the above queries efficiently, maintain two compound indexes, on:

1. (``discussion_id, posted)`` and
2. (``discussion_id, full_slug)``

Issue the following operation at the interactive Python shell.

```
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('posted', 1)])
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('full_slug', 1)])
```

Note: Ensure that you always sort by the final element in a compound index to maximize the performance of these queries.

Retrieve Comments via Direct Links

Queries

To directly retrieve a comment, without needing to page through all comments, you can select by the `slug` field:

```
comment = db.comments.find_one({
    'discussion_id': discussion_id,
    'slug': comment_slug})
```

You can retrieve a "sub-discussion," or a comment and all of its descendants recursively, by performing a regular expression prefix query on the `full_slug` field:

```
import re

subdiscussion = db.comments.find_one({
    'discussion_id': discussion_id,
    'full_slug': re.compile('^' + re.escape(parent_slug)) })
subdiscussion = subdiscussion.sort('full_slug')
```

Indexing

Since you have already created indexes on `{ discussion_id: 1, full_slug: }` to support retrieving sub-discussions, you can add support for the above queries by adding an index on `{ discussion_id: 1 , slug: 1 }`. Use the following operation in the Python shell:

```
>>> db.comments.ensure_index([
...     ('discussion_id', 1), ('slug', 1)])
```

28.2.3 Embedding All Comments

This design embeds the entire discussion of a comment thread inside of the topic *document*. In this example, the "topic," document holds the total content for whatever content you're managing.

Schema

Consider the following prototype `topic` document:

```
{
    _id: ObjectId(...),
    ... lots of topic data ...
    comments: [
        { posted: ISODateTime(...),
          author: { id: ObjectId(...), name: 'Rick' },
          text: 'This is so bogus ... ' },
       ... ]
}
```

This structure is only suitable for a chronological display of all comments because it embeds comments in chronological order. Each document in the array in the `comments` contains the comment's date, author, and text.

Note: Since you're storing the comments in sorted order, there is no need to maintain per-comment slugs.

To support threading using this design, you would need to embed comments within comments, using a structure that resembles the following:

```
{
    _id: ObjectId(...),
    ... lots of topic data ...
    replies: [
        { posted: ISODateTime(...),
          author: { id: ObjectId(...), name: 'Rick' },
          text: 'This is so bogus ... ',
          replies: [
              { author: { ... }, ... },
       ... ]
}
```

Here, the `replies` field in each comment holds the sub-comments, which can intern hold sub-comments.

Note: In the embedded document design, you give up some flexibility regarding display format, because it is difficult to display comments *except* as you store them in MongoDB.

If, in the future, you want to switch from chronological to threaded or from threaded to chronological, this design would make that migration quite expensive.

Warning: Remember that *BSON* documents have a *16 megabyte size limit* (page 679). If popular discussions grow larger than 16 megabytes, additional document growth will fail.
Additionally, when MongoDB documents grow significantly after creation you will experience greater storage fragmentation and degraded update performance while MongoDB migrates documents internally.

Operations

This section contains an overview of common operations for interacting with comments represented using a schema that embeds all comments the *document* of the "parent" or topic content.

Note: For all operations below, there is no need for any new indexes since all the operations are function within documents. Because you would retrieve these documents by the `_id` field, you can rely on the index that MongoDB creates automatically.

Post a new comment

To post a new comment in a chronologically ordered (i.e unthreaded) system, you need the following `update()`:

```
db.discussion.update(
    { 'discussion_id': discussion_id },
    { '$push': { 'comments': {
        'posted': datetime.utcnow(),
        'author': author_info,
        'text': comment_text } } } )
```

The `$push` operator inserts comments into the `comments` array in correct chronological order. For threaded discussions, the `update()` operation is more complex. To reply to a comment, the following code assumes that it can retrieve the 'path' as a list of positions, for the parent comment:

```
if path != []:
    str_path = '.'.join('replies.%d' % part for part in path)
    str_path += '.replies'
else:
    str_path = 'replies'
db.discussion.update(
    { 'discussion_id': discussion_id },
    { '$push': {
        str_path: {
            'posted': datetime.utcnow(),
            'author': author_info,
            'text': comment_text } } } )
```

This constructs a field name of the form `replies.0.replies.2...` as `str_path` and then uses this value with the `$push` operator to insert the new comment into the parent comment's `replies` array.

View Paginated Comments

To view the comments in a non-threaded design, you must use the `$slice` (page 470) operator:

```
discussion = db.discussion.find_one(
    {'discussion_id': discussion_id},
    { ... some fields relevant to your page from the root discussion ...,
```

```
      'comments': { '$slice': [ page_num * page_size, page_size ] }
    })
```

To return paginated comments for the threaded design, you must retrieve the whole document and paginate the comments within the application:

```
discussion = db.discussion.find_one({'discussion_id': discussion_id})

def iter_comments(obj):
    for reply in obj['replies']:
        yield reply
        for subreply in iter_comments(reply):
            yield subreply

paginated_comments = itertools.slice(
    iter_comments(discussion),
    page_size * page_num,
    page_size * (page_num + 1))
```

Retrieve a Comment via Direct Links

Instead of retrieving comments via slugs as above, the following example retrieves comments using their position in the comment list or tree.

For chronological (i.e. non-threaded) comments, just use the `$slice` (page 470) operator to extract a comment, as follows:

```
discussion = db.discussion.find_one(
    {'discussion_id': discussion_id},
    {'comments': { '$slice': [ position, position ] } })
comment = discussion['comments'][0]
```

For threaded comments, you must find the correct path through the tree in your application, as follows:

```
discussion = db.discussion.find_one({'discussion_id': discussion_id})
current = discussion
for part in path:
    current = current.replies[part]
comment = current
```

Note: Since parent comments embed child replies, this operation actually retrieves the entire sub-discussion for the comment you queried for.

See Also:

`find_one()`.

28.2.4 Hybrid Schema Design

Schema

In the "hybrid approach" you will store comments in "buckets" that hold about 100 comments. Consider the following example document:

```
{
    _id: ObjectId(...),
    discussion_id: ObjectId(...),
    page: 1,
    count: 42,
    comments: [ {
        slug: '34db',
        posted: ISODateTime(...),
        author: { id: ObjectId(...), name: 'Rick' },
        text: 'This is so bogus ... ' },
    ... ]
}
```

Each document maintains `page` and `count` data that contains meta data regarding the page, the page number and the comment count, in addition to the `comments` array that holds the comments themselves.

Note: Using a hybrid format makes storing threaded comments complex, and this specific configuration is not covered in this document.

Also, 100 comments is a *soft* limit for the number of comments per page. This value is arbitrary: choose a value that will prevent the maximum document size from growing beyond the 16MB *BSON document size limit* (page 679), but large enough to ensure that most comment threads will fit in a single document. In some situations the number of comments per document can exceed 100, but this does not affect the correctness of the pattern.

Operations

This section contains a number of common operations that you may use when building a CMS using this hybrid storage model with documents that hold 100 comment "pages."

All examples in this document use the Python programming language and the PyMongo *driver* for MongoDB, but you can implement this system using any language you choose.

Post a New Comment

Updating In order to post a new comment, you need to `$push` the comment onto the last page and `$inc` that page's comment count. Consider the following example that queries on the basis of a `discussion_id` field:

```
page = db.comment_pages.find_and_modify(
    { 'discussion_id': discussion['_id'],
      'page': discussion['num_pages'] },
    { '$inc': { 'count': 1 },
      '$push': {
          'comments': { 'slug': slug, ... } } },
    fields={'count':1},
    upsert=True,
    new=True )
```

The `find_and_modify()` operation is an *upsert*,: if MongoDB cannot find a document with the correct `page` number, the `find_and_modify()` will create it and initialize the new document with appropriate values for `count` and `comments`.

To limit the number of comments per page to roughly 100, you will need to create new pages as they become necessary. Add the following logic to support this:

```
if page['count'] > 100:
    db.discussion.update(
        { 'discussion_id: discussion['_id'],
          'num_pages': discussion['num_pages'] },
        { '$inc': { 'num_pages': 1 } } )
```

This `update()` operation includes the last known number of pages in the query to prevent a race condition where the number of pages incriments twice, that would result result in a nearly or totally empty document. If another process increments the number of pages, then update above does nothing.

Indexing To support the `find_and_modify()` and `update()` operations, maintain a compound index on (`discussion_id`, `page`) in the `comment_pages` collection, by issuing the following operation at the Python/PyMongo console:

```
>>> db.comment_pages.ensure_index([
...     ('discussion_id', 1), ('page', 1)])
```

View Paginated Comments

The following function defines how to paginate comments with a *fixed* page size (i.e. not with the roughly 100 comment documents in the above example,) as en example:

```
def find_comments(discussion_id, skip, limit):
    result = []
    page_query = db.comment_pages.find(
        { 'discussion_id': discussion_id },
        { 'count': 1, 'comments': { '$slice': [ skip, limit ] } })
    page_query = page_query.sort('page')
    for page in page_query:
        result += page['comments']
        skip = max(0, skip - page['count'])
        limit -= len(page['comments'])
        if limit == 0: break
    return result
```

Here, the `$slice` (page 470) operator pulls out comments from each page, but *only* when this satisfies the `skip` requirement. For example: if you have 3 pages with 100, 102, 101, and 22 comments on each page, and you wish to retrieve comments where `skip=300` and `limit=50`. Use the following algorithm:

Skip	Limit	Discussion
300	50	`{$slice: [ 300, 50 ] }` matches nothing in page #1; subtract page #1's `count` from `skip` and continue.
200	50	`{$slice: [ 200, 50 ] }` matches nothing in page #2; subtract page #2's `count` from `skip` and continue.
98	50	`{$slice: [ 98, 50 ] }` matches 2 comments in page #3; subtract page #3's `count` from `skip` (saturating at 0), subtract 2 from limit, and continue.
0	48	`{$slice: [ 0, 48 ] }` matches all 22 comments in page #4; subtract 22 from `limit` and continue.
0	26	There are no more pages; terminate loop.

Note: Since you already have an index on (`discussion_id`, `page`) in your `comment_pages` collection, MongoDB can satisfy these queries efficiently.

Retrieve a Comment via Direct Links

Query To retrieve a comment directly without paging through all preceding pages of commentary, use the slug to find the correct page, and then use application logic to find the correct comment:

```
page = db.comment_pages.find_one(
    { 'discussion_id': discussion_id,
      'comments.slug': comment_slug},
    { 'comments': 1 })
for comment in page['comments']:
    if comment['slug'] = comment_slug:
        break
```

Indexing To perform this query efficiently you'll need a new index on the `discussion_id` and `comments.slug` fields (i.e. `{ discussion_id: 1 comments.slug: 1 }`.) Create this index using the following operation in the Python/PyMongo console:

```
>>> db.comment_pages.ensure_index([
...     ('discussion_id', 1), ('comments.slug', 1)])
```

28.2.5 Sharding

For all of the architectures discussed above, you will want to the `discussion_id` field to participate in the shard key, if you need to shard your application.

For applications that use the "one document per comment" approach, consider using `slug` (or `full_slug`, in the case of threaded comments) fields in the shard key to allow the `mongos` (page 676) instances to route requests by `slug`. Issue the following operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'comments', {
...     'key' : { 'discussion_id' : 1, 'full_slug': 1 } })
```

This will return the following response:

```
{ "collectionsharded" : "comments", "ok" : 1 }
```

In the case of comments that fully-embedded in parent content *documents* the determination of the shard key is outside of the scope of this document.

For hybrid documents, use the page number of the comment page in the shard key along with the `discussion_id` to allow MongoDB to split popular discussions between, while grouping discussions on the same shard. Issue the following operation at the Python/PyMongo console:

```
>>> db.command('shardCollection', 'comment_pages', {
...     key : { 'discussion_id' : 1, 'page': 1 } })
{ "collectionsharded" : "comment_pages", "ok" : 1 }
```

CHAPTER

TWENTYNINE

PYTHON APPLICATION DEVELOPMENT

29.1 Write a Tumblelog Application with Django MongoDB Engine

29.1.1 Introduction

In this tutorial, you will learn how to create a basic tumblelog application using the popular Django Python web-framework and the *MongoDB* database.

The tumblelog will consist of two parts:

1. A public site that lets people view posts and comment on them.
2. An admin site that lets you add, change and delete posts and publish comments.

This tutorial assumes that you are already familiar with Django and have a basic familiarity with MongoDB operation and have *installed MongoDB* (page 9).

Where to get help

If you're having trouble going through this tutorial, please post a message to mongodb-user or join the IRC chat in *#mongodb* on irc.freenode.net to chat with other MongoDB users who might be able to help.

Note: Django MongoDB Engine uses a forked version of Django 1.3 that adds non-relational support.

29.1.2 Installation

Begin by installing packages required by later steps in this tutorial.

Prerequisite

This tutorial uses pip to install packages and virtualenv to isolate Python environments. While these tools and this configuration are not required as such, they ensure a standard environment and are strongly recommended. Issue the following commands at the system prompt:

```
pip install virtualenv
virtualenv myproject
```

Respectively, these commands: install the `virtualenv` program (using `pip`) and create a isolated python environment for this project (named `myproject`.)

To activate `myproject` environment at the system prompt, use the following commands:

```
source myproject/bin/activate
```

Installing Packages

Django MongoDB Engine directly depends on:

- Django-nonrel, a fork of Django 1.3 that adds support for non-relational databases
- djangotoolbox, a bunch of utilities for non-relational Django applications and backends

Install by issuing the following commands:

```
pip install https://bitbucket.org/wkornewald/django-nonrel/get/tip.tar.gz
pip install https://bitbucket.org/wkornewald/djangotoolbox/get/tip.tar.gz
pip install https://github.com/django-nonrel/mongodb-engine/tarball/master
```

Continue with the tutorial to begin building the "tumblelog" application.

29.1.3 Build a Blog to Get Started

In this tutorial you will build a basic blog as the foundation of this application and use this as the basis of your tumblelog application. You will add the first post using the shell and then later use the Django administrative interface.

Call the `startproject` command, as with other Django projects, to get started and create the basic project skeleton:

```
django-admin.py startproject tumblelog
```

Configuring Django

Configure the database in the `tumblelog/settings.py` file:

```
DATABASES = {
   'default': {
      'ENGINE': 'django_mongodb_engine',
      'NAME': 'my_tumble_log'
   }
}
```

See Also:

The Django MongoDB Engine Settings documentation for more configuration options.

Define the Schema

The first step in writing a tumblelog in Django is to define the "models" or in MongoDB's terminology *documents*.

In this application, you will define posts and comments, so that each `Post` can contain a list of `Comments`. Edit the `tumblelog/models.py` file so it resembles the following:

```
from django.db import models
from django.core.urlresolvers import reverse

from djangotoolbox.fields import ListField, EmbeddedModelField


class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True, db_index=True)
    title = models.CharField(max_length=255)
    slug = models.SlugField()
    body = models.TextField()
    comments = ListField(EmbeddedModelField('Comment'), editable=False)

    def get_absolute_url(self):
        return reverse('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    class Meta:
        ordering = ["-created_at"]


class Comment(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    body = models.TextField(verbose_name="Comment")
    author = models.CharField(verbose_name="Name", max_length=255)
```

The Django "nonrel" code looks the same as vanilla Django, however there is no built in support for some of MongoDB's native data types like Lists and Embedded data. `djangotoolbox` handles these definitions.

See Also:

The Django MongoDB Engine fields documentation for more.

The models declare an index to the `Post` class. One for the `created_at` date as our frontpage will order by date: there is no need to add `db_index` on `SlugField` because there is a default index on `SlugField`.

Add Data with the Shell

The `manage.py` provides a shell interface for the application that you can use to insert data into the tumblelog. Begin by issuing the following command to load the Python shell:

```
python manage.py shell
```

Create the first post using the following sequence of operations:

```
>>> from tumblelog.models import *
>>> post = Post(
... title="Hello World!",
... slug="hello-world",
... body = "Welcome to my new shiny Tumble log powered by MongoDB and Django-MongoDB!"
... )
>>> post.save()
```

Add comments using the following sequence of operations:

```
>>> post.comments
[]
```

```
>>> comment = Comment(
... author="Joe Bloggs",
... body="Great post! I'm looking forward to reading your blog")
>>> post.comments.append(comment)
>>> post.save()
```

Finally, inspect the post:

```
>>> post = Post.objects.get()
>>> post
<Post: Hello World!>
>>> post.comments
[<Comment: Comment object>]
```

Add the Views

Because django-mongodb provides tight integration with Django you can use generic views to display the frontpage and post pages for the tumblelog. Insert the following content into the `urls.py` file to add the views:

```
from django.conf.urls.defaults import patterns, include, url
from django.views.generic import ListView, DetailView
from tumblelog.models import Post

urlpatterns = patterns('',
    url(r'^$', ListView.as_view(
        queryset=Post.objects.all(),
        context_object_name="posts_list"),
        name="home"
    ),
    url(r'^post/(?P<slug>[a-zA-Z0-9-]+)/$', PostDetailView.as_view(
        queryset=Post.objects.all(),
        context_object_name="post"),
        name="post"
    ),
)
```

Add Templates

In the tumblelog directory add the following directories `templates` and `templates/tumblelog` for storing the tumblelog templates:

```
mkdir -p templates/tumblelog
```

Configure Django so it can find the templates by updating `TEMPLATE_DIRS` in the `settings.py` file to the following:

```
import os.path
TEMPLATE_DIRS = (
    os.path.join(os.path.realpath(__file__), '../templates'),
)
```

Then add a base template that all others can inherit from. Add the following to `templates/base.html`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
```

```
    <meta charset="utf-8">
    <title>My Tumblelog</title>
    <link href="http://twitter.github.com/bootstrap/1.4.0/bootstrap.css" rel="stylesheet">
    <style>.content {padding-top: 80px;}</style>
  </head>

  <body>

    <div class="topbar">
      <div class="fill">
        <div class="container">
          <h1><a href="/" class="brand">My Tumblelog</a>! <small>Starring MongoDB and Django-MongoDB
        </div>
      </div>
    </div>

    <div class="container">
      <div class="content">
        {% block page_header %}{% endblock %}
        {% block content %}{% endblock %}
      </div>
    </div>

  </body>
</html>
```

Create the frontpage for the blog, which should list all the posts. Add the following template to the `templates/tumblelog/post_list.html`:

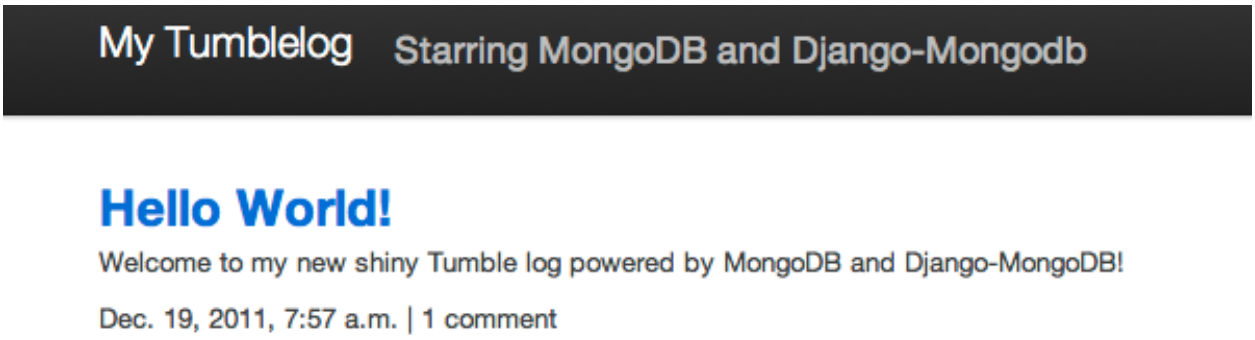
```
{% extends "base.html" %}

{% block content %}
    {% for post in posts_list %}
      <h2><a href="{% url post slug=post.slug %}">{{ post.title }}</a></h2>
      <p>{{ post.body|truncatewords:20 }}</p>
      <p>
        {{ post.created_at }} |
        {% with total=post.comments|length %}
            {{ total }} comment{{ total|pluralize }}
        {% endwith %}
      </p>
    {% endfor %}
{% endblock %}
```

Finally, add `templates/tumblelog/post_detail.html` for the individual posts:

```
{% extends "base.html" %}

{% block page_header %}
  <div class="page-header">
    <h1>{{ post.title }}</h1>
  </div>
{% endblock %}

{% block content %}
  <p>{{ post.body }}<p>
  <p>{{ post.created_at }}</p>
  <hr>
  <h2>Comments</h2>
```

```
  {% if post.comments %}
    {% for comment in post.comments %}
       <p>{{ comment.body }}</p>
       <p><strong>{{ comment.author }}</strong> <small>on {{ comment.created_at }}</small></p>
      {{ comment.text }}
    {% endfor %}
  {% endif %}
{% endblock %}
```

Run `python manage.py runserver` to see your new tumblelog! Go to http://localhost:8000/ and you should see:



29.1.4 Add Comments to the Blog

In the next step you will provide the facility for readers of the tumblelog to comment on posts. This a requires custom form and view to handle the form, and data. You will also update the template to include the form.

Create the Comments Form

You must customize form handling to deal with embedded comments. By extending `ModelForm`, it is possible to append the comment to the post on save. Create and add the following to `forms.py`:

```
from django.forms import ModelForm
from tumblelog.models import Comment


class CommentForm(ModelForm):

    def __init__(self, object, *args, **kwargs):
        """Override the default to store the original document
        that comments are embedded in.
        """
        self.object = object
        return super(CommentForm, self).__init__(*args, **kwargs)

    def save(self, *args):
        """Append to the comments list and save the post"""
        self.object.comments.append(self.instance)
        self.object.save()
        return self.object
```

```
    class Meta:
        model = Comment
```

Handle Comments in the View

You must extend the generic views need to handle the form logic. Add the following to the `views.py` file:

```
from django.http import HttpResponseRedirect
from django.views.generic import DetailView
from tumblelog.forms import CommentForm


class PostDetailView(DetailView):
    methods = ['get', 'post']

    def get(self, request, *args, **kwargs):
        self.object = self.get_object()
        form = CommentForm(object=self.object)
        context = self.get_context_data(object=self.object, form=form)
        return self.render_to_response(context)

    def post(self, request, *args, **kwargs):
        self.object = self.get_object()
        form = CommentForm(object=self.object, data=request.POST)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(self.object.get_absolute_url())

        context = self.get_context_data(object=self.object, form=form)
        return self.render_to_response(context)
```

Note: The `PostDetailView` class extends the `DetailView` class so that it can handle `GET` and `POST` requests. On `POST`, `post()` validates the comment: if valid, `post()` appends the comment to the post.

Don't forget to update the `urls.py` file and import the `PostDetailView` class to replace the `DetailView` class.

Add Comments to the Templates

Finally, you can add the form to the templates, so that readers can create comments. Splitting the template for the forms out into `templates/_forms.html` will allow maximum reuse of forms code:

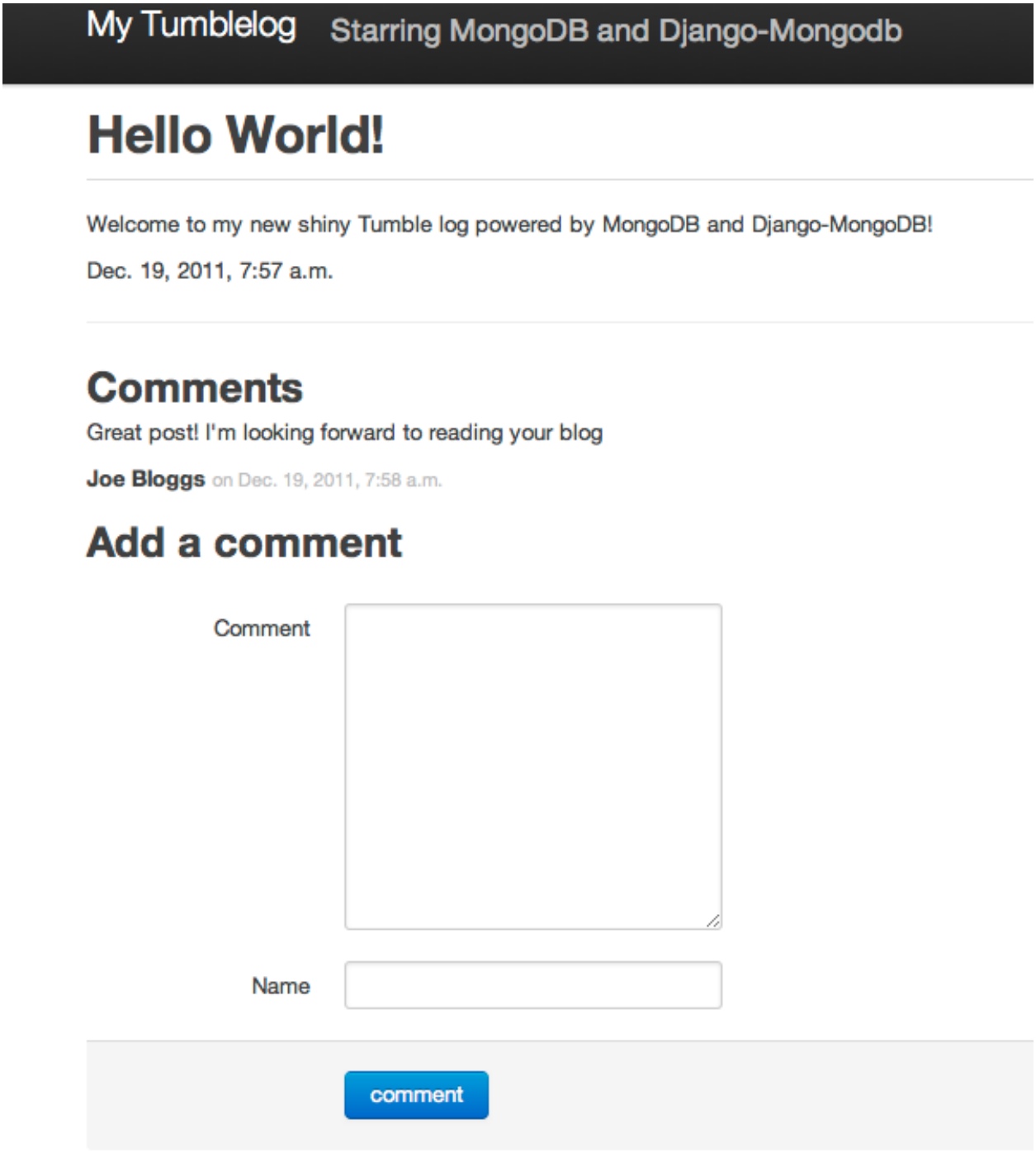
```
<fieldset>
{% for field in form.visible_fields %}
<div class="clearfix {% if field.errors %}error{% endif %}">
  {{ field.label_tag }}
  <div class="input">
    {{ field }}
    {% if field.errors or field.help_text %}
      <span class="help-inline">
      {% if field.errors %}
        {{ field.errors|join:' ' }}
      {% else %}
        {{ field.help_text }}
      {% endif %}
```

```
      </span>
    {% endif %}
  </div>
</div>
{% endfor %}
{% csrf_token %}
<div style="display:none">{% for h in form.hidden_fields %} {{ h }}{% endfor %}</div>
</fieldset>
```

After the comments section in `post_detail.html` add the following code to generate the comments form:

```
<h2>Add a comment</h2>
<form action="." method="post">
  {% include "_forms.html" %}
  <div class="actions">
    <input type="submit" class="btn primary" value="comment">
  </div>
</form>
```

Your tumblelog's readers can now comment on your posts! Run `python manage.py runserver` to see the changes. Run `python manage.py runserver` and go to http://localhost:8000/hello-world/ to see the following:



29.1.5 Add Site Administration Interface

While you may always add posts using the shell interface as above, you can easily create an administrative interface for posts with Django. Enable the admin by adding the following apps to `INSTALLED_APPS` in `settings.py`.

- `django.contrib.admin`
- `djangomongodbengine`
- `djangotoolbox`

- tumblelog

Warning: This application does not require the Sites framework. As a result, remove django.contrib.sites from INSTALLED_APPS. If you need it later please read SITE_ID issues document.

Create a admin.py file and register the Post model with the admin app:

```
from django.contrib import admin
from tumblelog.models import Post

admin.site.register(Post)
```

Note: The above modifications deviate from the default django-nonrel and djangotoolbox mode of operation. Django's administration module will not work unless you exclude the comments field. By making the comments field non-editable in the "admin" model definition, you will allow the administrative interface to function.

If you need an administrative interface for a ListField you must write your own Form / Widget.

See Also:

The Django Admin documentation docs for additional information.

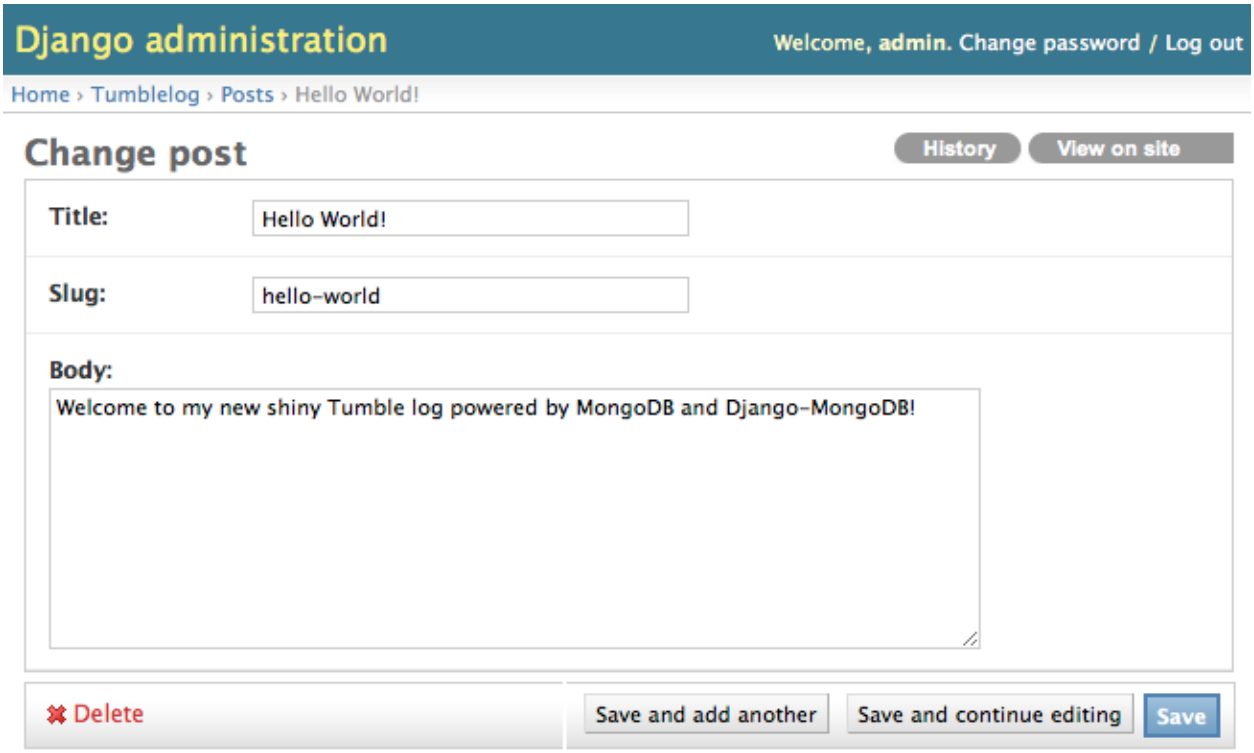
Update the urls.py to enable the administrative interface. Add the import and discovery mechanism to the top of the file and then add the admin import rule to the urlpatterns:

```
# Enable admin
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',

    # ...

    url(r'^admin/', include(admin.site.urls)),
)
```

Finally, add a superuser and setup the indexes by issuing the following command at the system prompt:

```
python manage.py syncdb
```

Once done run the server and you can login to admin by going to http://localhost:8000/admin/.



29.1.6 Convert the Blog to a Tumblelog

Currently, the application only supports posts. In this section you will add special post types including: *Video*, *Image* and *Quote* to provide a more traditional tumblelog application. Adding this data requires no migration.

In `models.py` update the `Post` class to add new fields for the new post types. Mark these fields with `blank=True` so that the fields can be empty.

Update `Post` in the `models.py` files to resemble the following:

```
POST_CHOICES = (
    ('p', 'post'),
    ('v', 'video'),
    ('i', 'image'),
    ('q', 'quote'),
)


class Post(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)
    title = models.CharField(max_length=255)
    slug = models.SlugField()

    comments = ListField(EmbeddedModelField('Comment'), editable=False)

    post_type = models.CharField(max_length=1, choices=POST_CHOICES, default='p')

    body = models.TextField(blank=True, help_text="The body of the Post / Quote")
    embed_code = models.TextField(blank=True, help_text="The embed code for video")
    image_url = models.URLField(blank=True, help_text="Image src")
    author = models.CharField(blank=True, max_length=255, help_text="Author name")
```

```
    def get_absolute_url(self):
        return reverse('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title
```

Note: Django-Nonrel doesn't support multi-table inheritance. This means that you will have to manually create an administrative form to handle data validation for the different post types.

The "Abstract Inheritance" facility means that the view logic would need to merge data from multiple collections.

The administrative interface should now handle adding multiple types of post. To conclude this process, you must update the frontend display to handle and output the different post types.

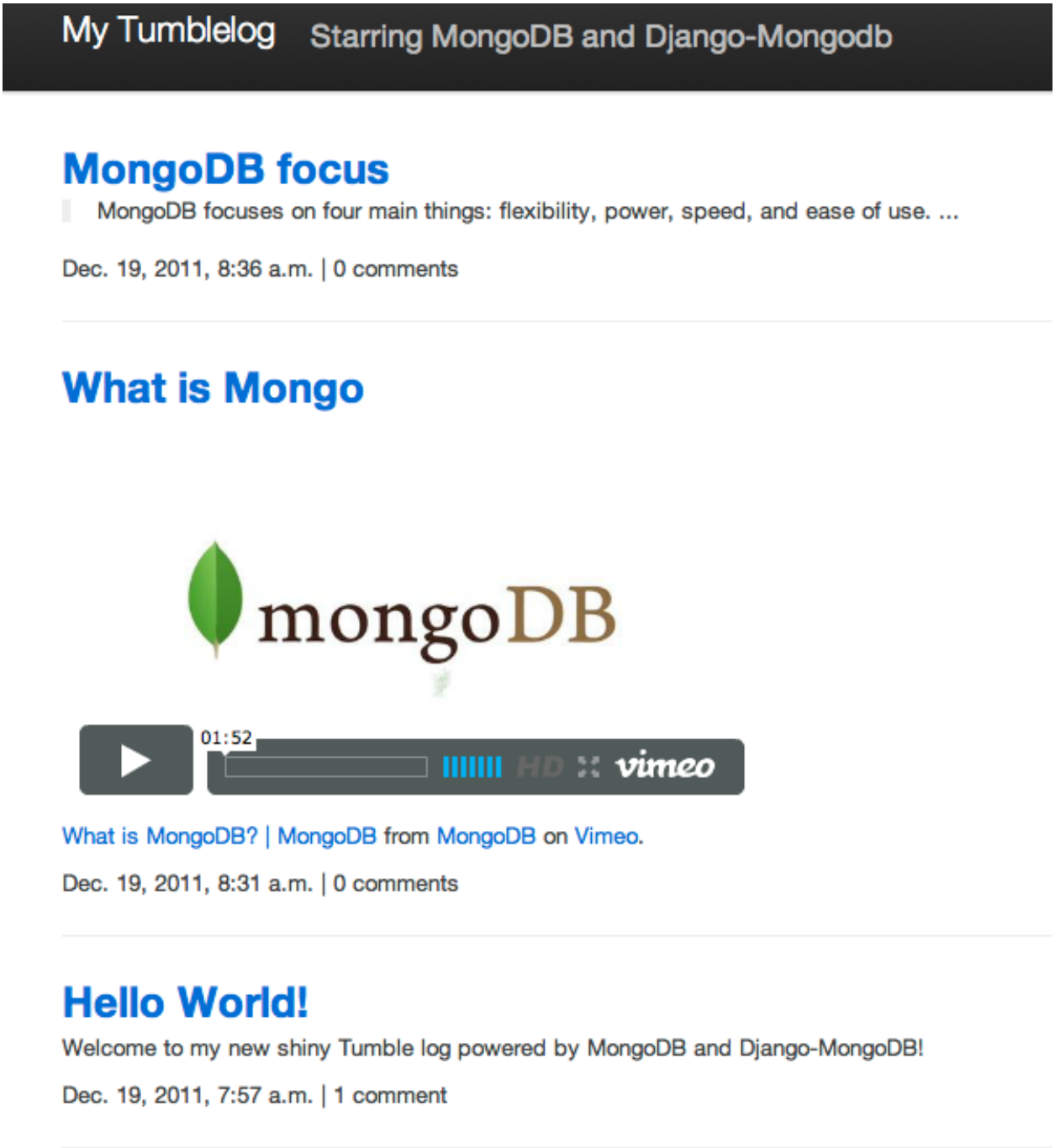
In the `post_list.html` file, change the post output display to resemble the following:

```
{% if post.post_type == 'p' %}
  <p>{{ post.body|truncatewords:20 }}</p>
{% endif %}
{% if post.post_type == 'v' %}
  {{ post.embed_code|safe }}
{% endif %}
{% if post.post_type == 'i' %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
{% if post.post_type == 'q' %}
  <blockquote>{{ post.body|truncatewords:20 }}</blockquote>
  <p>{{ post.author }}</p>
{% endif %}
```

In the `post_detail.html` file, change the output for full posts:

```
{% if post.post_type == 'p' %}
  <p>{{ post.body }}<p>
{% endif %}
{% if post.post_type == 'v' %}
  {{ post.embed_code|safe }}
{% endif %}
{% if post.post_type == 'i' %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
{% if post.post_type == 'q' %}
  <blockquote>{{ post.body }}</blockquote>
  <p>{{ post.author }}</p>
{% endif %}
```

Now you have a fully fledged tumbleblog using Django and MongoDB!



29.2 Write a Tumblelog Application with Flask and MongoEngine

29.2.1 Introduction

This tutorial describes the process for creating a basic tumblelog application using the popular Flask Python web-framework in conjunction with the *MongoDB* database.

The tumblelog will consist of two parts:

1. A public site that lets people view posts and comment on them.
2. An admin site that lets you add and change posts.

This tutorial assumes that you are already familiar with Flask and have a basic familiarity with MongoDB and have *installed MongoDB* (page 9). This tutorial uses MongoEngine as the Object Document Mapper (ODM,) this component may simplify the interaction between Flask and MongoDB.

Where to get help

If you're having trouble going through this tutorial, please post a message to mongodb-user or join the IRC chat in *#mongodb* on irc.freenode.net to chat with other MongoDB users who might be able to help.

29.2.2 Installation

Begin by installing packages required by later steps in this tutorial.

Prerequisite

This tutorial uses pip to install packages and virtualenv to isolate Python environments. While these tools and this configuration are not required as such, they ensure a standard environment and are strongly recommended. Issue the following command at the system prompt:

```
pip install virtualenv
virtualenv myproject
```

Respectively, these commands: install the `virtualenv` program (using `pip`) and create a isolated python environment for this project (named `myproject`.)

To activate `myproject` environment at the system prompt, use the following command:

```
source myproject/bin/activate
```

Install Packages

Flask is a "microframework," because it provides a small core of functionality and is highly extensible. For the "tumblelog" project, this tutorial includes task and the following extension:

- WTForms provides easy form handling.
- Flask-MongoEngine provides integration between MongoEngine, Flask, and WTForms.
- Flask-Script for an easy to use development server

Install with the following commands:

```
pip install flask
pip install flask-script
pip install WTForms
pip install mongoengine
pip install flask_mongoengine
```

Continue with the tutorial to begin building the "tumblelog" application.

29.2.3 Build a Blog to Get Started

First, create a simple "bare bones" application. Make a directory named `tumblelog` for the project and then, add the following content into a file named `__init__.py`:

```
from flask import Flask
app = Flask(__name__)


if __name__ == '__main__':
    app.run()
```

Next, create the `manage.py` file. [1] Use this file to load additional Flask-scripts in the future. Flask-scripts provides a development server and shell:

```
# Set the path
import os, sys
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from flask.ext.script import Manager, Server
from tumblelog import app

manager = Manager(app)

# Turn on debugger by default and reloader
manager.add_command("runserver", Server(
    use_debugger = True,
    use_reloader = True,
    host = '0.0.0.0')
)

if __name__ == "__main__":
    manager.run()
```

You can run this application with a test server, by issuing the following command at the system prompt:

```
python manage.py runserver
```

There should be no errors, and you can visit http://localhost:5000/ in a web browser to view a page with a "404" message.

Configure MongoEngine and Flask

Install the Flask extension and add the configuration. Update `tumblelog/__init__.py` so that it resembles the following:

```
from flask import Flask
from flask.ext.mongoengine import MongoEngine

app = Flask(__name__)
app.config["MONGODB_DB"] = "my_tumble_log"
app.config["SECRET_KEY"] = "KeepThisS3cr3t"

db = MongoEngine(app)

if __name__ == '__main__':
    app.run()
```

[1] This concept will be familiar to users of Django.

See Also:

The MongoEngine Settings documentation for additional configuration options.

Define the Schema

The first step in writing a tumblelog in Flask is to define the "models" or in MongoDB's terminology *documents*.

In this application, you will define posts and comments, so that each `Post` can contain a list of `Comments`. Edit the `models.py` file so that it resembles the following:

```
import datetime
from flask import url_for
from tumblelog import db


class Post(db.Document):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    body = db.StringField(required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))

    def get_absolute_url(self):
        return url_for('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    meta = {
        'allow_inheritance': True,
        'indexes': ['-created_at', 'slug'],
        'ordering': ['-created_at']
    }


class Comment(db.EmbeddedDocument):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    body = db.StringField(verbose_name="Comment", required=True)
    author = db.StringField(verbose_name="Name", max_length=255, required=True)
```

As above, MongoEngine syntax is simple and declarative. If you have a Django background, the syntax may look familiar. This example defines indexes for `Post`: one for the `created_at` date as our frontpage will order by date and another for the individual post `slug`.

Add Data with the Shell

The `manage.py` provides a shell interface for the application that you can use to insert data into the tumblelog. Before configuring the "urls" and "views" for this application, you can use this interface to interact with your the tumblelog. Begin by issuing the following command to load the Python shell:

```
python manage.py shell
```

Create the first post using the following sequence of operations:

```
>>> from tumblelog.models import *
>>> post = Post(
```

```
... title="Hello World!",
... slug="hello-world",
... body="Welcome to my new shiny Tumble log powered by MongoDB, MongoEngine, and Flask"
... )
>>> post.save()
```

Add comments using the following sequence of operations:

```
>>> post.comments
[]
>>> comment = Comment(
... author="Joe Bloggs",
... body="Great post! I'm looking forward to reading your blog!"
... )
>>> post.comments.append(comment)
>>> post.save()
```

Finally, inspect the post:

```
>>> post = Post.objects.get()
>>> post
<Post: Hello World!>
>>> post.comments
[<Comment: Comment object>]
```

Add the Views

Using Flask's class-based views system allows you to produce `List` and `Detail` views for tumblelog posts. Add `views.py` and create a *posts* blueprint:

```
from flask import Blueprint, request, redirect, render_template, url_for
from flask.views import MethodView
from tumblelog.models import Post, Comment

posts = Blueprint('posts', __name__, template_folder='templates')


class ListView(MethodView):

    def get(self):
        posts = Post.objects.all()
        return render_template('posts/list.html', posts=posts)


class DetailView(MethodView):

    def get(self, slug):
        post = Post.objects.get_or_404(slug=slug)
        return render_template('posts/detail.html', post=post)


# Register the urls
posts.add_url_rule('/', view_func=ListView.as_view('list'))
posts.add_url_rule('/<slug>/', view_func=DetailView.as_view('detail'))
```

Now in `__init__.py` register the blueprint, avoiding a circular dependency by registering the blueprints in a method. Add the following code to the module:

```
def register_blueprints(app):
    # Prevents circular imports
    from tumblelog.views import posts
    app.register_blueprint(posts)

register_blueprints(app)
```

Add this method and method call to the main body of the module and not in the main block.

Add Templates

In the `tumblelog` directory add the `templates` and `templates/posts` directories to store the tumblelog templates:

```
mkdir -p templates/posts
```

Create a base template. All other templates will inherit from this template, which should exist in the `templates/base.html` file:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My Tumblelog</title>
    <link href="http://twitter.github.com/bootstrap/1.4.0/bootstrap.css" rel="stylesheet">
    <style>.content {padding-top: 80px;}</style>
  </head>

  <body>

    {%- block topbar -%}
    <div class="topbar">
      <div class="fill">
        <div class="container">
          <h2>
              <a href="/" class="brand">My Tumblelog</a> <small>Starring Flask, MongoDB and MongoEngi
          </h2>
        </div>
      </div>
    </div>
    {%- endblock -%}

    <div class="container">
      <div class="content">
        {% block page_header %}{% endblock %}
        {% block content %}{% endblock %}
      </div>
    </div>
    {% block js_footer %}{% endblock %}
  </body>
</html>
```

Continue by creating a landing page for the blog that will list all posts. Add the following to the `templates/posts/list.html` file:

```
{% extends "base.html" %}

{% block content %}
```

```
    {% for post in posts %}
      <h2><a href="{{ url_for('posts.detail', slug=post.slug) }}">{{ post.title }}</a></h2>
      <p>{{ post.body|truncate(100) }}</p>
      <p>
        {{ post.created_at.strftime('%H:%M %Y-%m-%d') }} |
        {% with total=post.comments|length %}
            {{ total }} comment {%- if total > 1 %}s{%- endif -%}
        {% endwith %}
      </p>
    {% endfor %}
{% endblock %}
```

Finally, add `templates/posts/detail.html` template for the individual posts:

```
{% extends "base.html" %}

{% block page_header %}
  <div class="page-header">
    <h1>{{ post.title }}</h1>
  </div>
{% endblock %}

{% block content %}
  <p>{{ post.body }}<p>
  <p>{{ post.created_at.strftime('%H:%M %Y-%m-%d') }}</p>
  <hr>
  <h2>Comments</h2>
  {% if post.comments %}
    {% for comment in post.comments %}
       <p>{{ comment.body }}</p>
       <p><strong>{{ comment.author }}</strong> <small>on {{ comment.created_at.strftime('%H:%M %Y-%m
      {{ comment.text }}
    {% endfor %}
  {% endif %}
{% endblock %}
```

At this point, you can run the `python manage.py runserver` command again to see your new tumblelog! Go to http://localhost:5000 to see something that resembles the following:

29.2.4 Add Comments to the Blog

In the next step you will provide the facility for readers of the tumblelog to comment on posts. To provide commenting, you will create a form using WTForms that will update the view to handle the form data and update the template to include the form.

Handle Comments in the View

Begin by updating and refactoring the `views.py` file so that it can handle the form. Begin by adding the `import` statement and the `DetailView` class to this file:

```
from flask.ext.mongoengine.wtf import model_form

...

class DetailView(MethodView):

    form = model_form(Comment, exclude=['created_at'])

    def get_context(self, slug):
        post = Post.objects.get_or_404(slug=slug)
        form = self.form(request.form)

        context = {
            "post": post,
            "form": form
        }
        return context

    def get(self, slug):
        context = self.get_context(slug)
        return render_template('posts/detail.html', **context)

    def post(self, slug):
        context = self.get_context(slug)
        form = context.get('form')

        if form.validate():
            comment = Comment()
            form.populate_obj(comment)

            post = context.get('post')
            post.comments.append(comment)
            post.save()

            return redirect(url_for('posts.detail', slug=slug))

        return render_template('posts/detail.html', **context)
```

Note: `DetailView` extends the default Flask `MethodView`. This code remains DRY by defining a *get_context* method to get the default context for both `GET` and `POST` requests. On `POST`, `post()` validates the comment: if valid, `post()` appends the comment to the post.

Add Comments to the Templates

Finally, you can add the form to the templates, so that readers can create comments. Create a macro for the forms in `templates/_forms.html` will allow you to reuse the form code:

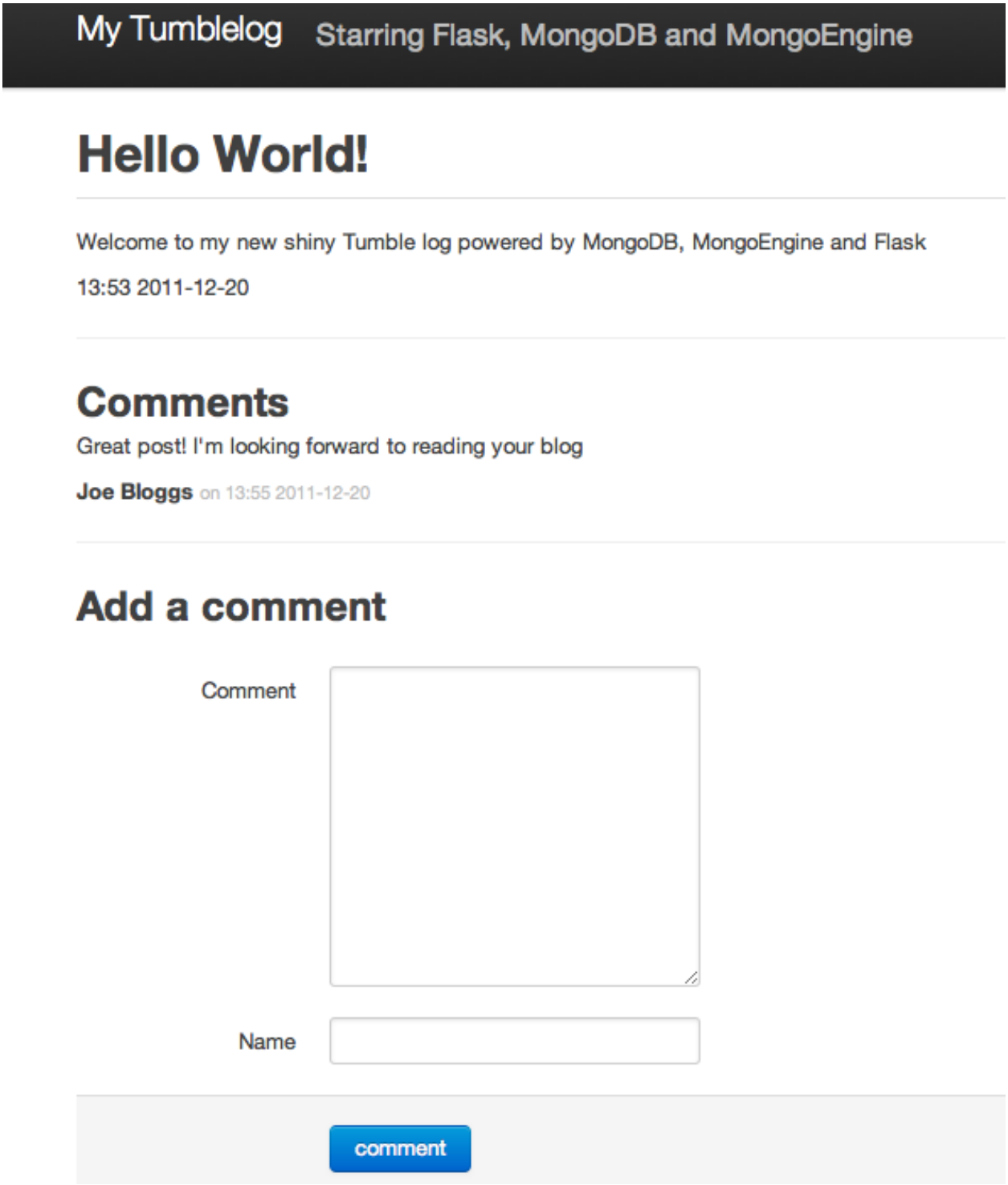
```
{% macro render(form) -%}
<fieldset>
{% for field in form %}
{% if field.type in ['CSRFTokenField', 'HiddenField'] %}
```

```
  {{ field() }}
{% else %}
  <div class="clearfix {% if field.errors %}error{% endif %}">
    {{ field.label }}
    <div class="input">
      {% if field.name == "body" %}
        {{ field(rows=10, cols=40) }}
      {% else %}
        {{ field() }}
      {% endif %}
      {% if field.errors or field.help_text %}
        <span class="help-inline">
        {% if field.errors %}
          {{ field.errors|join(' ') }}
        {% else %}
          {{ field.help_text }}
        {% endif %}
        </span>
      {% endif %}
    </div>
  </div>
{% endif %}
{% endfor %}
</fieldset>
{% endmacro %}
```

Add the comments form to `templates/posts/detail.html`. Insert an `import` statement at the top of the page and then output the form after displaying comments:

```
{% import "_forms.html" as forms %}

...

<hr>
<h2>Add a comment</h2>
<form action="." method="post">
  {{ forms.render(form) }}
  <div class="actions">
    <input type="submit" class="btn primary" value="comment">
  </div>
</form>
```

Your tumblelog's readers can now comment on your posts! Run `python manage.py runserver` to see the changes.



29.2.5 Add a Site Administration Interface

While you may always add posts using the shell interface as above, in this step you will add an administrative interface for the tumblelog site. To add the administrative interface you will add authentication and an additional view. This tutorial only addresses adding and editing posts: a "delete" view and detection of slug collisions are beyond the scope of this tutorial.

Add Basic Authentication

For the purposes of this tutorial all we need is a very basic form of authentication. The following example borrows from the an example Flask "Auth snippet". Create the file `auth.py` with the following content:

```
from functools import wraps
from flask import request, Response


def check_auth(username, password):
    """This function is called to check if a username /
    password combination is valid.
    """
    return username == 'admin' and password == 'secret'


def authenticate():
    """Sends a 401 response that enables basic auth"""
    return Response(
    'Could not verify your access level for that URL.\n'
    'You have to login with proper credentials', 401,
    {'WWW-Authenticate': 'Basic realm="Login Required"'})


def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            return authenticate()
        return f(*args, **kwargs)
    return decorated
```

Note: This creates a *requires_auth* decorator: provides basic authentication. Decorate any view that needs authentication with this decorator. The username is `admin` and password is `secret`.

Write an Administrative View

Create the views and admin blueprint in `admin.py`. The following view is deliberately generic, to facilitate customization.

```
from flask import Blueprint, request, redirect, render_template, url_for
from flask.views import MethodView

from flask.ext.mongoengine.wtf import model_form

from tumblelog.auth import requires_auth
from tumblelog.models import Post, Comment

admin = Blueprint('admin', __name__, template_folder='templates')


class List(MethodView):
    decorators = [requires_auth]
    cls = Post
```

```
    def get(self):
        posts = self.cls.objects.all()
        return render_template('admin/list.html', posts=posts)


class Detail(MethodView):

    decorators = [requires_auth]

    def get_context(self, slug=None):
        form_cls = model_form(Post, exclude=('created_at', 'comments'))

        if slug:
            post = Post.objects.get_or_404(slug=slug)
            if request.method == 'POST':
                form = form_cls(request.form, inital=post._data)
            else:
                form = form_cls(obj=post)
        else:
            post = Post()
            form = form_cls(request.form)

        context = {
            "post": post,
            "form": form,
            "create": slug is None
        }
        return context

    def get(self, slug):
        context = self.get_context(slug)
        return render_template('admin/detail.html', **context)

    def post(self, slug):
        context = self.get_context(slug)
        form = context.get('form')

        if form.validate():
            post = context.get('post')
            form.populate_obj(post)
            post.save()

            return redirect(url_for('admin.index'))
        return render_template('admin/detail.html', **context)


# Register the urls
admin.add_url_rule('/admin/', view_func=List.as_view('index'))
admin.add_url_rule('/admin/create/', defaults={'slug': None}, view_func=Detail.as_view('create'))
admin.add_url_rule('/admin/<slug>/', view_func=Detail.as_view('edit'))
```

Note: Here, the `List` and `Detail` views are similar to the frontend of the site; however, `requires_auth` decorates both views.

The "Detail" view is slightly more complex: to set the context, this view checks for a slug and if there is no slug, `Detail` uses the view for creating a new post. If a slug exists, `Detail` uses the view for editing an existing post.

In the `__init__.py` file update the `register_blueprints()` method to import the new admin blueprint.

```
def register_blueprints(app):
    # Prevents circular imports
    from tumblelog.views import posts
    from tumblelog.admin import admin
    app.register_blueprint(posts)
    app.register_blueprint(admin)
```

Create Administrative Templates

Similar to the user-facing portion of the site, the administrative section of the application requires three templates: a base template a list view, and a detail view.

Create an `admin` directory for the templates. Add a simple main index page for the admin in the `templates/admin/base.html` file:

```
{% extends "base.html" %}

{%- block topbar -%}
<div class="topbar" data-dropdown="dropdown">
  <div class="fill">
    <div class="container">
      <h2>
           <a href="{{ url_for('admin.index') }}" class="brand">My Tumblelog Admin</a>
      </h2>
      <ul class="nav secondary-nav">
        <li class="menu">
           <a href="{{ url_for("admin.create") }}" class="btn primary">Create new post</a>
        </li>
      </ul>
    </div>
  </div>
</div>
{%- endblock -%}
```

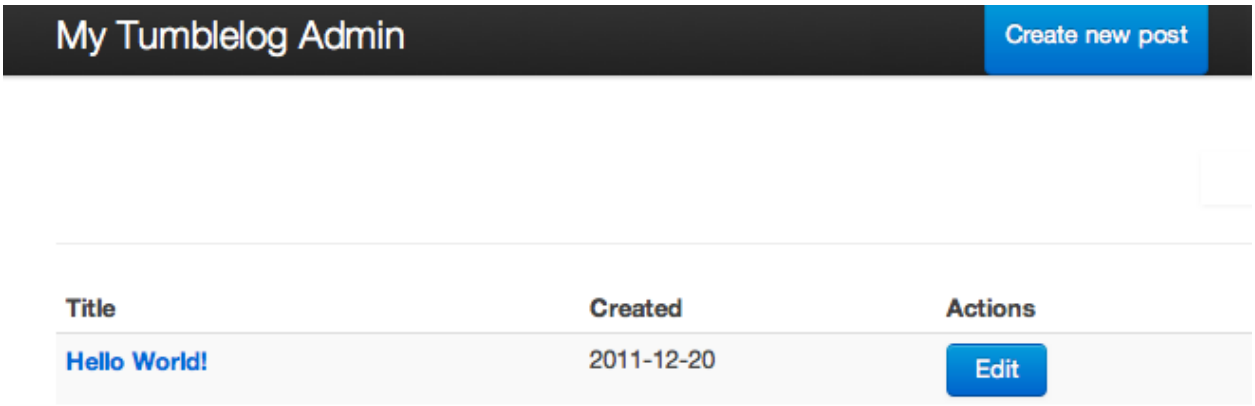
List all the posts in the `templates/admin/list.html` file:

```
{% extends "admin/base.html" %}

{% block content %}
  <table  class="condensed-table zebra-striped">
    <thead>
      <th>Title</th>
      <th>Created</th>
      <th>Actions</th>
    </thead>
    <tbody>
    {% for post in posts %}
      <tr>
        <th><a href="{{ url_for('admin.edit', slug=post.slug) }}">{{ post.title }}</a></th>
        <td>{{ post.created_at.strftime('%Y-%m-%d') }}</td>
        <td><a href="{{ url_for("admin.edit", slug=post.slug) }}" class="btn primary">Edit</a></td>
      </tr>
    {% endfor %}
    </tbody>
  </table>
{% endblock %}
```

Add a temple to create and edit posts in the `templates/admin/detail.html` file:

```
{% extends "admin/base.html" %}
{% import "_forms.html" as forms %}

{% block content %}
  <h2>
    {% if create %}
      Add new Post
    {% else %}
      Edit Post
    {% endif %}
  </h2>

  <form action="?{{ request.query_string }}" method="post">
    {{ forms.render(form) }}
    <div class="actions">
      <input type="submit" class="btn primary" value="save">
      <a href="{{ url_for("admin.index") }}" class="btn secondary">Cancel</a>
    </div>
  </form>
{% endblock %}
```

The administrative interface is ready for use. Restart the test server (i.e. `runserver`) so that you can log in to the administrative interface located at http://localhost:5000/admin/. (The username is `admin` and the password is `secret`.)



29.2.6 Converting the Blog to a Tumblelog

Currently, the application only supports posts. In this section you will add special post types including: *Video*, *Image* and *Quote* to provide a more traditional tumblelog application. Adding this data requires no migration because MongoEngine supports document inheritance.

Begin by refactoring the `Post` class to operate as a base class and create new classes for the new post types. Update the `models.py` file to include the code to replace the old `Post` class:

```
class Post(db.DynamicDocument):
    created_at = db.DateTimeField(default=datetime.datetime.now, required=True)
    title = db.StringField(max_length=255, required=True)
    slug = db.StringField(max_length=255, required=True)
    comments = db.ListField(db.EmbeddedDocumentField('Comment'))
```

```
    def get_absolute_url(self):
        return url_for('post', kwargs={"slug": self.slug})

    def __unicode__(self):
        return self.title

    @property
    def post_type(self):
        return self.__class__.__name__

    meta = {
        'allow_inheritance': True,
        'indexes': ['-created_at', 'slug'],
        'ordering': ['-created_at']
    }


class BlogPost(Post):
    body = db.StringField(required=True)


class Video(Post):
    embed_code = db.StringField(required=True)


class Image(Post):
    image_url = db.StringField(required=True, max_length=255)


class Quote(Post):
    body = db.StringField(required=True)
    author = db.StringField(verbose_name="Author Name", required=True, max_length=255)
```

Note: In the `Post` class the `post_type` helper returns the class name, which will make it possible to render the various different post types in the templates.

As MongoEngine handles returning the correct classes when fetching `Post` objects you do not need to modify the interface view logic: only modify the templates.

Update the `templates/posts/list.html` file and change the post output format as follows:

```
{% if post.body %}
  {% if post.post_type == 'Quote' %}
    <blockquote>{{ post.body|truncate(100) }}</blockquote>
    <p>{{ post.author }}</p>
  {% else %}
    <p>{{ post.body|truncate(100) }}</p>
  {% endif %}
{% endif %}
{% if post.embed_code %}
  {{ post.embed_code|safe() }}
{% endif %}
{% if post.image_url %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
```

In the `templates/posts/detail.html` change the output for full posts as follows:

```
{% if post.body %}
  {% if post.post_type == 'Quote' %}
    <blockquote>{{ post.body }}</blockquote>
    <p>{{ post.author }}</p>
  {% else %}
    <p>{{ post.body }}</p>
  {% endif %}
{% endif %}
{% if post.embed_code %}
  {{ post.embed_code|safe() }}
{% endif %}
{% if post.image_url %}
  <p><img src="{{ post.image_url }}" /><p>
{% endif %}
```

Updating the Administration

In this section you will update the administrative interface to support the new post types.

Begin by, updating the `admin.py` file to import the new document models and then update `get_context()` in the `Detail` class to dynamically create the correct model form to use:

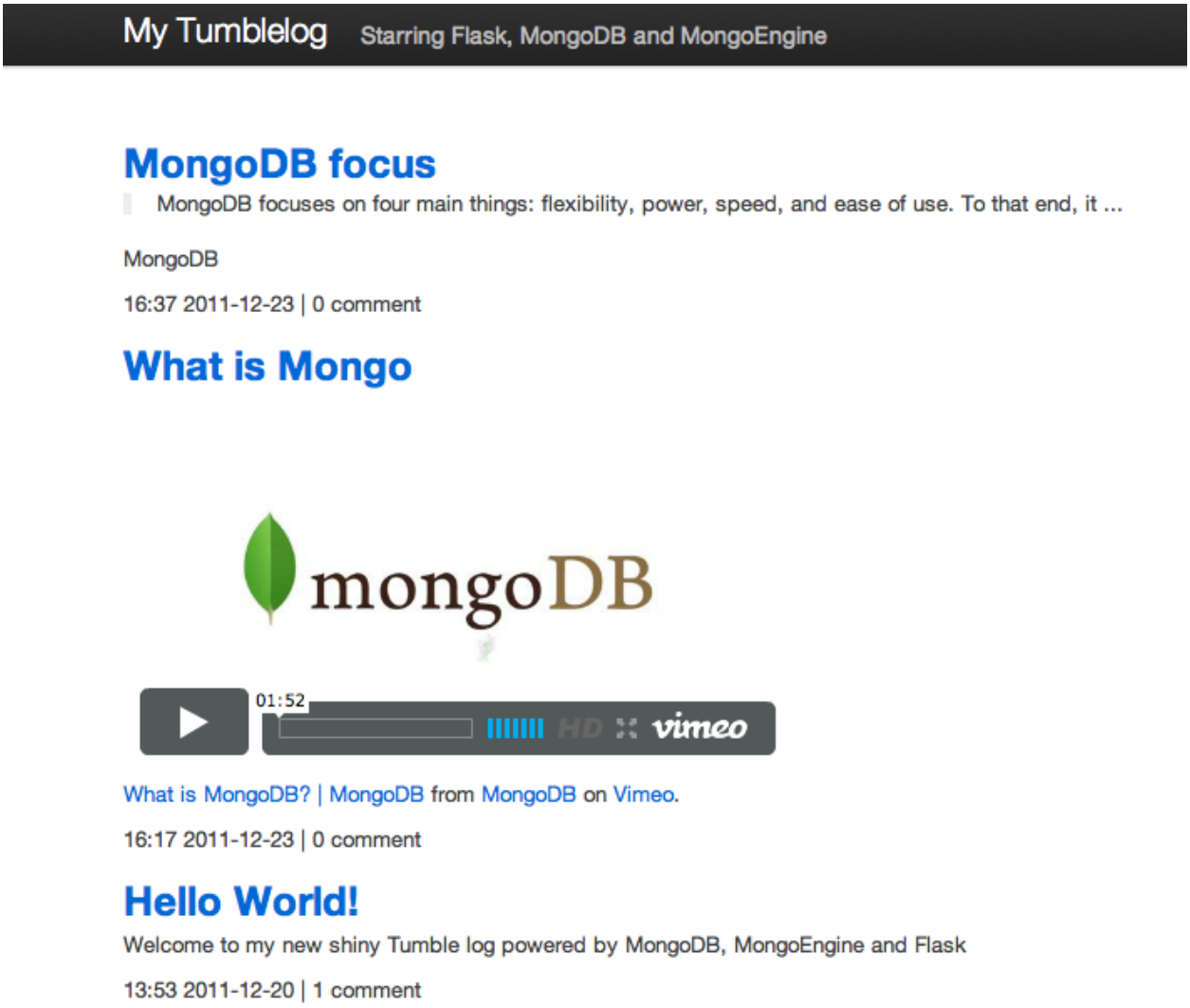
```
from tumblelog.models import Post, BlogPost, Video, Image, Quote, Comment

# ...

class Detail(MethodView):

    decorators = [requires_auth]
    # Map post types to models
    class_map = {
        'post': BlogPost,
        'video': Video,
        'image': Image,
        'quote': Quote,
    }

    def get_context(self, slug=None):

        if slug:
            post = Post.objects.get_or_404(slug=slug)
            # Handle old posts types as well
            cls = post.__class__ if post.__class__ != Post else BlogPost
            form_cls = model_form(cls,  exclude=('created_at', 'comments'))
            if request.method == 'POST':
                form = form_cls(request.form, inital=post._data)
            else:
                form = form_cls(obj=post)
        else:
            # Determine which post type we need
            cls = self.class_map.get(request.args.get('type', 'post'))
            post = cls()
            form_cls = model_form(cls,  exclude=('created_at', 'comments'))
            form = form_cls(request.form)
        context = {
            "post": post,
            "form": form,
```

```
            "create": slug is None
        }
        return context

    # ...
```

Update the `template/admin/base.html` file to create a new post drop down menu in the toolbar:

```
{% extends "base.html" %}

{%- block topbar -%}
<div class="topbar" data-dropdown="dropdown">
  <div class="fill">
    <div class="container">
      <h2>
           <a href="{{ url_for('admin.index') }}" class="brand">My Tumblelog Admin</a>
      </h2>
      <ul class="nav secondary-nav">
        <li class="menu">
          <a href="#" class="menu">Create new</a>
          <ul class="menu-dropdown">
            {% for type in ('post', 'video', 'image', 'quote') %}
                <li><a href="{{ url_for("admin.create", type=type) }}">{{ type|title }}</a></li>
            {% endfor %}
          </ul>
        </li>
      </ul>
    </div>
  </div>
</div>
{%- endblock -%}

{% block js_footer %}
  <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
  <script src="http://twitter.github.com/bootstrap/1.4.0/bootstrap-dropdown.js"></script>
{% endblock %}
```

Now you have a fully fledged tumbleblog using Flask and MongoEngine!



29.2.7 Additional Resources

The complete source code is available on Github: <https://github.com/rozza/flask-tumblelog>

Part XII

Frequently Asked Questions

CHAPTER

THIRTY

FAQ: MONGODB FUNDAMENTALS

This document answers basic questions about MongoDB.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 411) or post your question to the MongoDB User Mailing List.

Frequently Asked Questions:

30.1 What kind of Database is MongoDB?

MongoDB is *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*.

30.2 What languages can I use to work with the MongoDB?

MongoDB *client drivers* exist for all of the most popular programming languages, and many of the less popular ones. See the latest list of drivers for details.

See Also:

"*Drivers* (page 285)."

30.3 Does MongoDB support SQL?

No.

However, MongoDB does support a rich, ad-hoc query language of it's own.

See Also:

The query "`http://docs.mongodb.org/manual/reference/operators`" document and the Query Overview and the Tour pages from the wiki.

30.4 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 app, gaming, e-commerce, analytics, archiving, and logging.

Do not use MongoDB for systems that require SQL, joins, and multi-object transactions.

30.5 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 application requiring multi-object commit with rollback generally aren't feasible.

30.6 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 it's 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.

30.7 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.

30.8 Are writes written to disk immediately, or lazily?

Writes are physically written to the journal within 100 milliseconds. 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.

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. This does not affect durability, as the journal has enough information to ensure crash recovery.

30.9 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.

30.10 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.

30.11 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.

30.12 What are the 32-bit limitations?

MongoDB uses memory-mapped files. 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 Limitations

Note: 32-bit builds disable *journaling* by default because journaling further limits the maximum amount of data that the database can store.

CHAPTER

THIRTYONE

FAQ: MONGODB FOR APPLICATION DEVELOPERS

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 411) or post your question to the MongoDB User Mailing List.

Frequently Asked Questions:

31.1 What is a "namespace?"

A "namespace" is the concatenation of the *database* name and the *collection* names 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 in a single collection of data files.

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 the `acme.user.history` is a valid namespace, with the `acme` database name, and the `user.history` collection name.

31.2 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:

```
db.people.find().forEach( function(x){db.user.insert(x)} );
```

Note: Because this process decodes *BSON* documents to *JSON* during the copy procedure, documents you may incur a loss of type-fidelity.

Consider using `mongodump` and `mongorestore` to maintain type fidelity.

Also consider the `cloneCollection` *command* that may provide some of this functionality.

31.3 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use `db.collection.remove()`, the object will no longer exist in MongoDB's on-disk data storage.

31.4 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* every 100 milliseconds. These values are configurable with the `journalCommitInterval` (page 625) and `syncdelay` (page 627).

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 resents 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.

31.5 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 Atomic Operations wiki page.

31.6 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new "*aggregation framework* (page 253)," with the `aggregate` command.

MongoDB also supports *map-reduce* with the `mapReduce`, as well as basic aggregation with the `group`, `count`, and `distinct`. commands.

See Also:

The Aggregation wiki page.

31.7 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` (page 622) option or the command-line option `--quiet` (page 581) to suppress these messages from the log.

31.8 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:

The "MongoDB on the Amazon Platform" wiki page.

31.9 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` (page 627) flag to modify the file preallocation strategy.

See Also:

This wiki page that address MongoDB disk use.

31.10 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 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 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:

  ```
  { last_name : "Smith", best_score: 3.9 }
  ```

  If you shorten the filed 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.

31.11 How does MongoDB address SQL or Query injection?

31.11.1 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 285) 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.

31.11.2 JavaScript

All of the following MongoDB operations permit you to run arbitrary JavaScript expressions directly on the server:- `$where`:

- `$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.

31.11.3 Dollar Sign Operator Escaping

Field names in MongoDB's query language have a semantic. 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+FFOE` (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 `db.collection.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.

31.11.4 Driver-Specific Issues

See the "PHP MongoDB Driver Security Notes" page in the PHP driver documentation for more information

31.12 How does MongoDB provide concurrency?

MongoDB implements a server-wide reader-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.

A more granular approach to locking will appear in MongoDB v2.2. For now, several yielding optimizations exist to mitigate the coarseness of the lock. These include:

- Yielding on long operations. Queries and updates that operate on multiple document may yield to writers
- Yielding on page faults. If an update or query is likely to trigger a page fault, then the operation will yield to keep from blocking other clients for the duration of the page fault.

31.13 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 compare order:

- Null
- Numbers (ints, longs, doubles)
- Symbol, String
- Object
- Array
- BinData
- ObjectID
- Boolean
- Date, Timestamp
- Regular Expression

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Consider the following `mongo` example:

```
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("4b03155dce8de6586fb002c7"), "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 wiki page for an example of a C++ use of `MinKey`.
- The jsobj.h source file for the definition of `MinKey` and `MaxKey`.

31.14 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 driver.

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()` find method, in as the following:

```
db.getCollection("_foo").find()
```

CHAPTER

THIRTYTWO

FAQ: SHARDING WITH MONGODB

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 411) or post your question to the MongoDB User Mailing List.

Frequently Asked Questions:

32.1 Is sharding appropriate for a new deployment?

Sometimes.

If your data set fits on a single servers, 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.

32.2 How does sharding work with replication?

To use replication with sharding, deploy each *shard* as a *replica set*.

32.3 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* (page 116). This reality underscores the important of choosing a good *shard key* (page 109). 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 121) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See `shardCollection`, `sh.shardCollection()`, *Sharded Cluster Administration* (page 113), the *Shard Key* (page 133) section in the *Sharding Internals* (page 133) document, *Deploy a Sharded Cluster* (page 141), and SERVER-4000 for more information.

32.4 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.

32.5 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.

32.6 What happens if a client updates a document in a chunk during a migration?

The `mongos` (page 676) 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.

32.7 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` (page 676) will merely wait for the shard to return results.

32.8 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` (page 676) 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` (page 676) 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` (page 676) 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` (page 676) must parallelize the query over the shards and perform a merge-sort on the `email` of the documents found.

32.9 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` (page 676) merges each shard's results before returning them to the client.

32.10 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 works well in this case.

32.11 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 at least 64 MB 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.

32.12 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.

You can set `noMoveParanoia` (page 630) to `true` to disable this behavior.

32.13 How many connections does each `mongos` need?

Typically, `mongos` (page 676) uses one connection from each client, as well as one outgoing connection to each shard, or each member of the replica set that backs each shard. If you've enabled the `slaveOk` bit, then the mongos may create two or more connections per replica set.

32.14 Why does `mongos` hold connections?

`mongos` (page 676) 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` (page 676) with a large number open of connections. If the `mongos` (page 676) is no longer in use, you're safe restarting the process to close existing connections.

32.15 Where does MongoDB report on connections used by `mongos`?

Connect to the `mongos` (page 676) with the `mongo` shell, and run the following command:

```
db._adminCommand("connPoolStats");
```

32.16 What does `writebacklisten` in the log mean?

The writeback listener is a process that opens a long poll to detect non-safe writes sent to a server and to send them back to the correct server if necessary.

These messages are a key part of the sharding infrastructure and should not cause concern.

32.17 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.

32.18 What is the process for moving, renaming, or changing the number of config servers?

See Also:

The wiki page that describes this process: "Changing Configuration Servers."

32.19 When do the mongos servers detect config server changes?

mongos (page 676) 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 (page 676) 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 (page 676) to force it to refresh its cache.

32.20 Is it possible to quickly update mongos servers after updating a replica set configuration?

The mongos (page 676) instances will detect these changes without intervention over time. However, if you want to force the mongos (page 676) to reload its configuration, run the flushRouterConfig command against to each mongos (page 676) directly.

32.21 What does the maxConns setting on mongos do?

The maxConns (page 622) option limits the number of connections accepted by mongos (page 676).

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 (page 676) layer.

Set maxConns (page 622) 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 (page 676) from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

32.22 How do indexes impact queries in sharded systems?

If the query does not include the *shard key*, the mongos (page 676) 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` (page 676) 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 document for more information.

32.23 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.

32.24 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.

32.25 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 or read operations rather than inserts, this limitation should not impact your performance. However, if you have a high insert volume, this may be a limitation.

32.26 Can shard key be in ascending order, like dates or timestamps?

If you insert documents with monotonically increasing shard keys, all inserts 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 or read operations rather than inserts, this limitation should not impact your performance. However, if you have a high insert volume, a monotonically increasing shard key may be a limitation.

To address this issue, you can use a field with a value that stores the hash of a key with an ascending value. While you can compute a hashed value in your application and include this value in your documents for use as a shard key, the SERVER-2001 issue will implement this capability within MongoDB.

32.27 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 138), 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 Group or 10gen support to address this issue.

CHAPTER

THIRTYTHREE

FAQ: REPLICA SETS AND REPLICATION IN MONGODB

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 411) or post your question to the MongoDB User Mailing List.

Frequently Asked Questions:

33.1 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.

33.2 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 a 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.

33.3 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.

33.4 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.

33.5 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 Distributed Replica Set (page 79)

33.6 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 between 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* the set will automatically elect one of the *secondary* members of the set as primary.

33.7 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.

33.8 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.

33.9 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.

33.10 Are write operations durable without `getLastError`?

Yes.

However, if you want confirmation that a given write has arrived safely at the server, you must also run the `getLastError` command after each write. If you enable your driver's *write concern*, or "safe mode", the driver will automatically send `getLastError` this command. If you want to guarantee that a given write syncs to the journal, you must pass the `{j: true}` option `getLastError` (or specify it as part of the write concern).

33.11 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 original 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 51).

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:

Replication Architectures (page 51)

33.12 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 *Using MongoDB with SSL Connections* (page 167) for more information. Run all arbiters on secure networks, as with all MongoDB components.

See Also:

The overview of *Arbiter Members of Replica Sets* (page 41).

33.13 Which members of a replica set vote in elections?

All members of a replica set, unless the value of `votes` (page 664) is equal to `0`, vote in elections. This includes all *delayed* (page 40), *hidden* (page 40) and *secondary-only* (page 39) members, as well as the *arbiters* (page 41).

See Also:

Elections (page 34)

33.14 Do hidden members vote in replica set elections?

Hidden members (page 40) of term:*replica :sets do* vote in elections. To exclude a member from voting in an :election, change the value of the member's `votes :` (page 664) configuration to `0`.

See Also:

Elections (page 34)

33.15 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 preallocation can lead to some variation in storage utilization between nodes. Storage use disparities will be most pronounced when you add members at different times.

CHAPTER

THIRTYFOUR

FAQ: MONGODB STORAGE

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 411) or post your question to the MongoDB User Mailing List.

Frequently Asked Questions:

34.1 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 content 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.

34.2 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.

34.3 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.

34.4 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.

34.5 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.

34.6 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 *Database Statistics Reference* (page 651) document outlines the meaning of the fields in the `db.stats()` output.

34.7 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 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.

CHAPTER

THIRTYFIVE

FAQ: INDEXES

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 411) or post your question to the MongoDB User Mailing List. See also *Indexing Strategies* (page 242).

Frequently Asked Questions:

35.1 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 *Building Indexes on Replica Sets* (page 241).

35.2 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 driver.

35.3 How do you determine the size of an index?

To check the sizes of the indexes on a collection, use `db.collection.stats()`.

35.4 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 245).

35.5 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.

35.6 How do you determine what fields to index?

A number of factors determine what fields to index, including *selectivity* (page 245), 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 Strategies* (page 242).

35.7 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.

35.8 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 Options* (page 235).

If you build a large index without the background option, and if doing so causes the database to stop responding, wait for the index to finish building.

35.9 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 245). If you need to use these, it is often best to make sure that an additional, more selective criterion is part of the query.

Part XIII

Reference

CHAPTER

THIRTYSIX

MONGODB INTERFACE

36.1 Reference

36.1.1 Query, Update, Projection, and Aggregation Operators

- Query and update operators:

$addToSet

$addToSet

The `$addToSet` operator adds a value to an array only *if* the value is *not* in the array already. If the value *is* in the array, `$addToSet` returns without modifying the array. Otherwise, `$addToSet` behaves the same as `$push`. Consider the following example:

```
db.collection.update( { field: value }, { $addToSet: { field: value1 } } );
```

Here, `$addToSet` appends `value1` to the array stored in `field`, *only if* `value1` is not already a member of this array.

$each operator is only used with the `$addToSet` see the documentation of *$addToSet* (page 443) for more information.

$each

The `$each` is available within the `$addToSet`, which allows you to add multiple values to the array if they do not exist in the `field` array in a single operation. Consider the following prototype:

```
db.collection.update( { field: value }, { $addToSet: { field: { $each : [ value1, value2
```

$all

$all

Syntax: `{ field: { $all: [ <value> , <value1> ... ] } }`

`$all` selects the documents where the `field` holds an array and contains all elements (e.g. `<value>`, `<value1>`, etc.) in the array.

Consider the following example:

```
db.inventory.find( { tags: { $all: [ "appliances", "school", "book" ] } } )
```

This query selects all documents in the `inventory` collection where the `tags` field contains an array with the elements, `appliances`, `school`, and `technology`.

Therefore, the above query will match documents in the `inventory` collection that have a `tags` field that hold *either* of the following arrays:

```
[ "school", "book", "bag", "headphone", "appliances" ]
[ "appliances", "school", "book" ]
```

The `$all` operator exists to describe and specify arrays in MongoDB queries. However, you may use the `$all` operator to select against a non-array `field`, as in the following example:

```
db.inventory.find( { qty: { $all: [ 50 ] } } )
```

However, use the following form to express the same query:

```
db.inventory.find( { qty: 50 } )
```

Both queries will select all documents in the `inventory` collection where the value of the `qty` field equals `50`.

Note: In most cases, MongoDB does not treat arrays as sets. This operator provides a notable exception to this approach.

In the current release queries that use the `$all` operator must scan all the documents that match the first element in the query array. As a result, even with an index to support the query, the operation may be long running, particularly when the first element in the array is not very selective.

See Also:

`find()`, `update()`, and `$set`.

$and

`$and`

New in version 2.0. *Syntax*: `{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }`

`$and` performs a logical `AND` operation on an array of *two or more* expressions (e.g. `<expression1>`, `<expression2>`, etc.) and selects the documents that satisfy *all* the expressions in the array. The `$and` operator uses *short-circuit evaluation*. If the first expression (e.g. `<expression1>`) evaluates to `false`, MongoDB will not evaluate the remaining expressions.

Consider the following example:

```
db.inventory.find({ $and: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

- `price` field value equals `1.99` **and**
- `qty` field value is less than `20` **and**
- `sale` field value is equal to `true`.

MongoDB provides an implicit `AND` operation when specifying a comma separated list of expressions. For example, you may write the above query as:

```
db.inventory.find( { price: 1.99, qty: { $lt: 20 } , sale: true } )
```

If, however, a query requires an AND operation on the same field such as `{ $price: { $ne: 1.99 } } AND { price: { $exists: true } }`, then either use the `$and` operator for the two separate expressions or combine the operator expressions for the field `{ $price: { $ne: 1.99, $exists: true } }`.

Consider the following examples:

```
db.inventory.update( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] }, {

db.inventory.update( { price: { $ne: 1.99, $exists: true } } , { $set: { qty: 15 } } )
```

Both `update()` operations will set the value of the `qty` field in documents where:

–the `price` field value does not equal `1.99` **and**

–the `price` field exists.

See Also:

`find()`, `update()`, `$ne`, `$exists`, `$set`.

$atomic

`$atomic`

In multi-update mode, it's possible to specify an `$atomic` "operator" that allows you to **isolate** some updates from each other within this operation. Consider the following example:

```
db.foo.update( { field1 : 1 , $atomic : 1 }, { $inc : { field2 : 1 } } , false , true )
```

Without the `$atomic` operator, multi-updates will allow other operations to interleave with this updates. If these interleaved operations contain writes, the update operation may produce unexpected results. By specifying `$atomic` you can guarantee isolation for the entire multi-update.

See Also:

See `db.collection.update()` for more information about the `db.collection.update()` method.

$bit

`$bit`

The `$bit` operator performs a bitwise update of a field. Only use this with integer fields. For example:

```
db.collection.update( { field: 1 }, { $bit: { field: { and: 5 } } } );
```

Here, the `$bit` operator updates the integer value of the field named `field` with a bitwise `and: 5` operation. This operator only works with number types.

$box

`$box`

New in version 1.4. The `$box` operator specifies a rectangular shape for the `$within` operator in *geospatial* queries. To use the `$box` operator, you must specify the bottom left and top right corners of the rectangle in an array object. Consider the following example:

```
db.collection.find( { loc: { $within: { $box: [ [0,0], [100,100] ] } } } )
```

This will return all the documents that are within the box having points at: `[0,0]`, `[0,100]`, `[100,0]`, and `[100,100]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$center

`$center`

New in version 1.4. This specifies a circle shape for the `$within` operator in *geospatial* queries. To define the bounds of a query using `$center`, you must specify:

–the center point, and

–the radius

Considering the following example:

```
db.collection.find( { location: { $within: { $center: [ [0,0], 10 ] } } } );
```

The above command returns all the documents that fall within a 10 unit radius of the point `[0,0]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$centerSphere

`$centerSphere`

New in version 1.8. The `$centerSphere` operator is the spherical equivalent of the `$center` operator. `$centerSphere` uses spherical geometry to calculate distances in a circle specified by a point and radius.

Considering the following example:

```
db.collection.find( { loc: { $centerSphere: { [0,0], 10 / 3959 } } } )
```

This query will return all documents within a 10 mile radius of `[0,0]` using a spherical geometry to calculate distances.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$comment

`$comment`

The `$comment` (page 446) makes it possible to attach a comment to a query. Because these comments propagate to the `profile` log, adding `$comment` (page 446) modifiers can make your profile data much easier to interpret and trace. Consider the following example:

```
db.collection.find()._addSpecial( "$comment" , "[COMMENT]" )
```

Here, `[COMMENT]` represents the text of the comment.

$each

Note: The `$each` operator is only used with the `$addToSet` see the documentation of *$addToSet* (page 443) for more information.

`$each`

The `$each` is available within the `$addToSet`, which allows you to add multiple values to the array if they do not exist in the `field` array in a single operation. Consider the following prototype:

```
db.collection.update( { field: value }, { $addToSet: { field: { $each : [ value1, value2, va
```

$elemMatch (query)

See Also:

$elemMatch (projection) (page 469)

`$elemMatch`

New in version 1.4. The `$elemMatch` (page 469) operator matches more than one component within an array element. For example,

```
db.collection.find( { array: { $elemMatch: { value1: 1, value2: { $gt: 1 } } } } );
```

returns all documents in `collection` where the array `array` satisfies all of the conditions in the `$elemMatch` (page 469) expression, or where the value of `value1` is 1 and the value of `value2` is greater than 1. Matching arrays must have at least one element that matches all specified criteria. Therefore, the following document would not match the above query:

```
{ array: [ { value1:1, value2:0 }, { value1:2, value2:2 } ] }
```

while the following document would match this query:

```
{ array: [ { value1:1, value2:0 }, { value1:1, value2:2 } ] }
```

$exists

`$exists`

Syntax: `{ field: { $exists: boolean } }`

`$exists` selects the documents that contain the field. MongoDB *$exists* does **not** correspond to SQL operator `exists`. For SQL `exists`, refer to the `$in` operator.

Consider the following example:

```
db.inventory.find( { qty: { $exists: true, $nin: [ 5, 15 ] } } )
```

This query will select all documents in the `inventory` collection where the `qty` field exists *and* its value does not equal either `5` nor `15`.

See Also:

`find()`, `$and`, `$nin`, `$in`.

$explain

$explain

Use the $explain (page 448) operator to return a *document* that describes the process and indexes used to return the query. This may provide useful insight when attempting to optimize a query. Consider the following example:

```
db.collection.find()._addSpecial( "$explain", 1 )
```

The JavaScript function `cursor.explain()` provides equivalent functionality in the `mongo` shell. See the following example, which is equivalent to the above:

```
db.collection.find().explain()
```

$gt

$gt

Syntax: `{field: {$gt: value} }`

`$gt` selects those documents where the value of the `field` is greater than (i.e. >) the specified `value`.

Consider the following example:

```
db.inventory.find( { qty: { $gt: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is greater than `20`.

Consider the following example which uses the `$gt` operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` operation will set the value of the `price` field in the documents that contain the embedded document `carrier` whose `fee` field value is greater than `2`.

See Also:

`find()`, `update()`, `$set`.

$gte

$gte

Syntax: `{field: {$gte: value} }`

`$gte` selects the documents where the value of the `field` is greater than or equal to (i.e. >=) a specified value (e.g. `value`.)

Consider the following example:

```
db.inventory.find( { qty: { $gte: 20 } } )
```

This query would select all documents in `inventory` where the `qty` field value is greater than or equal to `20`.

Consider the following example which uses the `$gte` operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` operation will set the value of the `price` field that contain the embedded document `carrier` whose‘‘fee‘‘ field value is greater than or equal to `2`.

See Also:

`find()`, `update()`, `$set`.

$hint

`$hint`

Use the `$hint` (page 449) operator to force the query optimizer to use a specific index to fulfill the query. Use `$hint` (page 449) for testing query performance and indexing strategies. Consider the following form:

```
db.collection.find()._addSpecial( "$hint", { _id : 1 } )
```

This operation returns all documents in the collection named `collection` using the index on the `_id` field. Use this operator to override MongoDB's default index selection process and pick indexes manually.

$in

`$in`

Syntax: `{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }`

`$in` selects the documents where the `field` value equals any value in the specified array (e.g. `<value1>`, `<value2>`, etc.)

Consider the following example:

```
db.inventory.find( { qty: { $in: [ 5, 15 ] } } )
```

This query will select to select all documents in the `inventory` collection where the `qty` field value is either `5` or `15`. Although you can express this query using the `$or` operator, choose the `$in` operator rather than the `$or` operator when performing equality checks on the same field.

If the `field` holds an array, then the `$in` operator selects the documents whose `field` holds an array that contains at least one element that matches a value in the specified array (e.g. `<value1>`, `<value2>`, etc.)

Consider the following example:

```
db.inventory.update( { tags: { $in: ["appliances", "school"] } }, { $set: { sale:true } } )
```

This `update()` operation will set the `sale` field value in the `inventory` collection where the `tags` field holds an array with at least one element matching an element in the array `["appliances", "school"]`.

See Also:

`find()`, `update()`, `$or`, `$set`.

$inc

`$inc`

The `$inc` operator increments a value by a specified amount if field is present in the document. If the field does not exist, `$inc` sets field to the number value. For example:

```
db.collection.update( { field: value }, { $inc: { field1: amount } } );
```

In this example, for documents in `collection` where `field` has the value `value`, the value of `field1` increments by the value of `amount`. The above operation only increments the *first* matching document *unless* you specify multi-update:

```
db.collection.update( { age: 20 }, { $inc: { age: 1 } } );
db.collection.update( { name: "John" }, { $inc: { age: 1 } } );
```

In the first example all documents that have an `age` field with the value of `20`, the operation increases `age` field by one. In the second example, in all documents where the `name` field has a value of `John` the operation increases the value of the `age` field by one.

`$inc` accepts positive and negative incremental amounts.

$lt

$lt

Syntax: `{field: {$lt: value} }`

`$lt` selects the documents where the value of the `field` is less than (i.e. <) the specified `value`.

Consider the following example:

```
db.inventory.find( { qty: { $lt: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is less than `20`.

Consider the following example which uses the `$lt` operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )
```

This `update()` operation will set the `price` field value in the documents that contain the embedded document `carrier` whose `fee` field value is less than `20`.

See Also:

`find()`, `update()`, `$set`.

$lte

$lte

Syntax: `{ field: { $lte: value} }`

`$lte` selects the documents where the value of the `field` is less than or equal to (i.e. <=) the specified `value`.

Consider the following example:

```
db.inventory.find( { qty: { $lte: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value is less than or equal to `20`.

Consider the following example which uses the `$lt` operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } } )
```

This `update()` operation will set the `price` field value in the documents that contain the embedded document `carrier` whose `fee` field value is less than or equal to `5`.

See Also:

`find()`, `update()`, `$set`.

$max

`$max`

Specify a `$max` (page 451) value to specify an upper boundary for the value of a field. `mongod` enforces this boundary with an index of that field.

```
db.collection.find()._addSpecial("$max" , { value : 100 })
```

This operation above limits the documents returned to those that match the query described by `[QUERY]` where the field `value` is less than `20`. `mongod` infers the index based on on the `query` unless specified by the `cursor.hint()` function.

Use operation alone or in conjunction with `$min` (page 452) to limit results to a specific range.

Note: In most cases, you should avoid this operator in favor of `$lt`.

$maxDistance

`$maxDistance`

The `$maxDistance` operator specifies an upper bound to limit the results of a geolocation query. See below, where the `$maxDistance` operator narrows the results of the `$near` query:

```
db.collection.find( { location: { $near: [100,100], $maxDistance: 10 } } );
```

This query will return documents with `location` fields from `collection` that have values with a distance of 5 or fewer units from the point `[100,100]`. `$near` returns results ordered by their distance from `[100,100]`. This operation will return the first 100 results unless you modify the query with the `cursor.limit()` method.

Specify the value of the `$maxDistance` argument in the same units as the document coordinate system.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$maxScan

`$maxScan`

Constrains the query to only scan the specified number of documents when fulfilling the query. Use the following form:

```
db.collection.find()._addSpecial( "$maxScan" , 50 )
```

Use this modifier to prevent potentially long running queries from disrupting performance by scanning through too much data.

$min

$min

Specify a $min (page 452) value to specify a lower boundary for the value of a field. mongod enforces this boundary with an index of the field.

```
db.collection.find( { [QUERY] } )._addSpecial("$min" , { value : 20})
```

This operation above limits the documents returned to those that match the query described by [QUERY] where the field value is at least 20. mongod infers the index based on the query unless specified by the cursor.hint() function.

Use operation alone or in conjunction with $max (page 451) to limit results to a specific range.

Note: In most cases, you should avoid this operator in favor of $gte.

$mod

$mod

Syntax: { field: { $mod: [divisor, remainder]} }

$mod selects the documents where the field value divided by the divisor has the specified remainder.

Consider the following example:

```
db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )
```

This query will select all documents in the inventory collection where the qty field value modulo 4 equals 3, such as documents with qty value equal to 0 or 12.

In some cases, you can query using the $mod operator rather than the more expensive $where operator. Consider the following example using the $mod operator:

```
db.inventory.find( { qty: { $mod: [ 4, 3 ] } } )
```

The above query is less expensive than the following query which uses the $where operator:

```
db.inventory.find( { $where: "this.qty % 4 == 3" } )
```

See Also:

find(), update(), $set.

$ne

$ne

Syntax: {field: {$ne: value} }

$ne selects the documents where the value of the field is not equal (i.e. !=) to the specified value. This includes documents that do not contain the field.

Consider the following example:

```
db.inventory.find( { qty: { $ne: 20 } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value does not equal `20`, including those documents that do not contain the `qty` field.

Consider the following example which uses the `$ne` operator with a field from an embedded document:

```
db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )
```

This `update()` operation will set the `qty` field value in the documents that contains the embedded document `carrier` whose `state` field value does not equal "NY", or where the `state` field or the `carrier` embedded document does not exist.

See Also:

`find()`, `update()`, `$set`.

$near

$near

The `$near` operator takes an argument, coordinates in the form of `[x, y]`, and returns a list of objects sorted by distance from those coordinates. See the following example:

```
db.collection.find( { location: { $near: [100,100] } } );
```

This query will return 100 ordered records with a `location` field in `collection`. Specify a different limit using the `cursor.limit()`, or another *geolocation operator*, or a non-geospatial operator to limit the results of the query.

Note: Specifying a batch size (i.e. `batchSize()` (page 529)) in conjunction with queries that use the `$near` is not undefined. See SERVER-5236 for more information.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$nearSphere

$nearSphere

New in version 1.8. The `$nearSphere` operator is the spherical equivalent of the `$near` operator. `$nearSphere` returns all documents near a point, calculating distances using spherical geometry.

```
db.collection.find( { loc: { $nearSphere: [0,0] } } )
```

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$nin

$nin

Syntax: `{ field: { $nin: [ <value1>, <value2> ... <valueN> ]} }`

`$nin` selects the documents where:

–the `field` value is not in the specified `array` **or**

–the `field` does not exist.

Consider the following query:

```
db.inventory.find( { qty: { $nin: [ 5, 15 ] } } )
```

This query will select all documents in the `inventory` collection where the `qty` field value does **not** equal `5` nor `15`. The selected documents will include those documents that do *not* contain the `qty` field.

If the `field` holds an array, then the `$nin` operator selects the documents whose `field` holds an array with **no** element equal to a value in the specified array (e.g. `<value1>`, `<value2>`, etc.).

Consider the following query:

```
db.inventory.update( { tags: { $nin: [ "appliances", "school" ] } }, { $set: { sale: false }
```

This `update()` operation will set the `sale` field value in the `inventory` collection where the `tags` field holds an array with **no** elements matching an element in the array `["appliances", "school"]` or where a document does not contain the `tags` field.

See Also:

`find()`, `update()`, `$set`.

$nor

$nor

Syntax: `{ $nor: [ { <expression1> }, { <expression2> }, ... { <expressionN> } ] }`

`$nor` performs a logical `NOR` operation on an array of *two or more* `<expressions>` and selects the documents that **fail** all the `<expressions>` in the array.

Consider the following example:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

–the `price` field value does *not* equal `1.99` **and**

–the `qty` field value is *not* less than `20` **and**

–the `sale` field value is *not* equal to `true`

including those documents that do not contain these field(s).

The exception in returning documents that do not contain the field in the `$nor` expression is when the `$nor` operator is used with the `$exists` operator.

Consider the following query which uses only the `$nor` operator:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { sale: true } ]  } )
```

This query will return all documents that:

–contain the `price` field whose value is *not* equal to `1.99` and contain the `sale` field whose value *is not* equal to `true` **or**

–contain the `price` field whose value is *not* equal to `1.99` *but* do *not* contain the `sale` field **or**

–do *not* contain the `price` field *but* contain the `sale` field whose value *is not* equal to `true` **or**

–do *not* contain the `price` field *and* do *not* contain the `sale` field

Compare that with the following query which uses the `$nor` operator with the `$exists` operator:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { price: { $exists: false } },
                             { sale: true }, { sale: { $exists: false } } ] } )
```

This query will return all documents that:

–contain the `price` field whose value is *not* equal to `1.99` and contain the `sale` field whose value *is not* equal to `true`

See Also:

`find()`, `update()`, `$set`, `$exists`.

$not

$not

Syntax: `{ field: { $not: { <operator-expression> } } }`

`$not` performs a logical `NOT` operation on the specified `<operator-expression>` and selects the documents that do *not* match the `<operator-expression>`. This includes documents that do not contain the `field`.

Consider the following query:

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

This query will select all documents in the `inventory` collection where:

–the `price` field value is less than or equal to `1.99` **or**

–the `price` field does not exist

`{ $not: { $gt: 1.99 } }` is different from the `$lte` operator. `{ $lt: 1.99 }` returns *only* the documents where `price` field exists and its value is less than or equal to `1.99`.

Remember that the `$not` operator only affects *other operators* and cannot check fields and documents independently. So, use the `$not` operator for logical disjunctions and the `$ne` operator to test the contents of fields directly.

Consider the following behaviors when using the `$not` operator:

–The operation of the `$not` operator is consistent with the behavior of other operators but may yield unexpected results with some data types like arrays.

–The `$not` operator does **not** support operations with the `$regex` operator. Instead use `http://docs.mongodb.org/manual/` or in your driver interfaces, use your language's regular expression capability to create regular expression objects.

Consider the following example which uses the pattern match expression `http://docs.mongodb.org/manual/`:

```
db.inventory.find( { item: { $not: /^p.*/ } } )
```

The query will select all documents in the `inventory` collection where the `item` field value does *not* start with the letter `p`.

If using PyMongo's `re.compile()`, you can write the above query as:

```
import re
for noMatch in db.inventory.find( { "item": { "$not": re.compile("^p.*") } } ):
    print noMatch
```

See Also:

`find()`, `update()`, `$set`, `$gt`, `$regex`, PyMongo, *driver*.

$or

$or

New in version 1.6.Changed in version 2.0: You may nest `$or` operations; however, these expressions are not as efficiently optimized as top-level. *Syntax*: `{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }`

The `$or` operator performs a logical `OR` operation on an array of *two or more* `<expressions>` and selects the documents that satisfy *at least* one of the `<expressions>`.

Consider the following query:

```
db.inventory.find( { price:1.99, $or: [ { qty: { $lt: 20 } }, { sale: true } ] } )
```

This query will select all documents in the `inventory` collection where:

–the `price` field value equals `1.99` *and*

–either the `qty` field value is less than `20` **or** the `sale` field value is `true`.

Consider the following example which uses the `$or` operator to select fields from embedded documents:

```
db.inventory.update( { $or: [ { price:10.99 }, { "carrier.state": "NY"} ] }, { $set: { sale:
```

This `update()` operation will set the value of the `sale` field in the documents in the `inventory` collection where:

–the `price` field value equals `10.99` **or**

–the `carrier` embedded document contains a field `state` whose value equals `NY`.

When using `$or` with `<expressions>` that are equality checks for the value of the same field, choose the `$in` operator over the `$or` operator.

Consider the query to select all documents in the `inventory` collection where:

–either `price` field value equals `1.99` *or* the `sale` field value equals `true`, **and**

–either `qty` field value equals `20` *or* `qty` field value equals `50`,

The most effective query would be:

```
db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ], qty: { $in: [20, 50] } } )
```

Consider the following behaviors when using the `$or` operator:

–When using indexes with `$or` queries, remember that each clause of an `$or` query will execute in parallel. These clauses can each use their own index. Consider the following query:

```
db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ] } )
```

For this query, you would create one index on `price` (`db.inventory.ensureIndex( { price: 1 } )`) and another index on `sale` (`db.inventory.ensureIndex( { sale: 1 } )`) rather than a compound index.

–Also, when using the `$or` operator with the `sort()` method in a query, the query will **not** use the indexes on the `$or` fields. Consider the following query which adds a `sort()` method to the above query:

```
db.inventory.find ( { $or: [ { price: 1.99 }, { sale: true } ] } ).sort({item:1})
```

This modified query will not use the index on `price` nor the index on `sale`.

See Also:

`find()`, `update()`, `$set`, `$and`, `sort()`.

$orderby

$orderby

The `$orderby` (page 457) operator sorts the results of a query in ascending or descending order. Consider the following syntax:

```
db.collection.find()._addSpecial( "$orderby", { age : -1} )
```

This is equivalent to the following `cursor.sort()` method that may be more familiar to you:

```
db.collection.find().sort( { age: -1 } )
```

Both of these examples return all documents in the collection named `collection` sorted for in descending order from greatest to smallest. Specify a value to `$orderby` (page 457) of negative one (e.g. `-1`, as above) to sort in descending order or a positive value (e.g. `1`) to sort in ascending order.

Unless you have a index for the specified key pattern, use `$orderby` (page 457) in conjunction with `$maxScan` (page 451) and/or `cursor.limit()` to avoid requiring MongoDB to perform a large in-memory sort. `cursor.limit()` increases the speed and reduce the amount of memory required to return this query by way of an optimized algorithm.

$polygon

$polygon

New in version 1.9. Use `$polygon` to specify a polygon for a bounded query using the `$within` operator for *geospatial* queries. To define the polygon, you must specify an array of coordinate points, as in the following:

```
[ [ x1,y1 ], [x2,y2], [x3,y3] ]
```

The last point specified is always implicitly connected to the first. You can specify as many points, and therefore sides, as you like. Consider the following bounded query for documents with coordinates within a polygon:

```
db.collection.find( { loc: { $within: { $polygon: [ [0,0], [3,6], [6,0]  ] } } } )
```

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$pop

$pop

The `$pop` operator removes the first or last element of an array. Pass `$pop` a value of `1` ‘ to remove the last element in an array and a value of `-1` to remove the first element of an array. Consider the following syntax:

```
db.collection.update( {field: value }, { $pop: { field: 1 } } );
```

This operation removes the last item of the array in `field` in the document that matches the query statement `{ field: value }`. The following example removes the *first* item of the same array:

```
db.collection.update( {field: value }, { $pop: { field: -1 } } );
```

Be aware of the following `$pop` behaviors:

- The `$pop` operation fails if `field` is not an array.
- `$pop` will successfully remove the last item in an array. `field` will then hold an empty array.

New in version 1.1.

$

$

Syntax: `{ "<array>.$" : value }`

The positional `$` operator identifies an element in an `array` field to update without explicitly specifying the position of the element in the array. The positional `$` operator, when used with the `update()` method and acts as a placeholder for the **first match** of the update `query selector`:

```
db.collection.update( { <query selector> }, { <update operator>: { "array.$" : value } } )
```

The `array` field **must** appear as part of the `query selector`.

Consider the following collection `students` with the following documents:

```
{ "_id" : 1, "grades" : [ 80, 85, 90 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

To update `80` to `82` in the `grades` array in the first document, use the positional `$` operator if you do not know the position of the element in the array:

```
db.students.update( { _id: 1, grades: 80 }, { $set: { "grades.$" : 82 } } )
```

Remember that the positional `$` operator acts as a placeholder for the **first match** of the update `query selector`.

The positional `$` operator facilitates updates to arrays that contain embedded documents. Use the positional `$` operator to access the fields in the embedded documents with the dot notation on the `$` operator.

```
db.collection.update( { <query selector> }, { <update operator>: { "array.$.field" : value }
```

Consider the following document in the `students` collection whose `grades` field value is an array of embedded documents:

```
{ "_id" : 4, "grades" : [ { grade: 80, mean: 75, std: 8 },
                          { grade: 85, mean: 90, std: 5 },
                          { grade: 90, mean: 85, std: 3 } ] }
```

Use the positional `$` operator to update the value of the `std` field in the embedded document with the `grade` of `85`:

```
db.students.update( { _id: 4, "grades.grade": 85 }, { $set: { "grades.$.std" : 6 } } )
```

Consider the following behaviors when using the positional `$` operator:

– Do not use the positional operator `$` with *upsert* operations because, inserts will use the `$` as a field name in the inserted document.

– When used with the `$unset` operator, the positional `$` operator does not remove the matching element from the array but rather sets it to `null`.

See Also:

`update()`, `$set` and `$unset`

$pull

$pull

The `$pull` operator removes all instances of a value from an existing array. Consider the following example:

```
db.collection.update( { field: value }, { $pull: { field: value1 } } );
```

`$pull` removes the value `value1` from the array in `field`, in the document that matches the query statement `{ field: value }` in `collection`. If `value1` existed multiple times in the `field` array, `$pull` would remove all instances of `value1` in this array.

$pullAll

$pullAll

The `$pullAll` operator removes multiple values from an existing array. `$pullAll` provides the inverse operation of the `$pushAll` operator. Consider the following example:

```
db.collection.update( { field: value }, { $pullAll: { field1: [ value1, value2, value3 ] } } );
```

Here, `$pullAll` removes `[ value1, value2, value3 ]` from the array in `field1`, in the document that matches the query statement `{ field: value }` in `collection`.

$push

$push

The `$push` operator appends a specified value to an array. For example:

```
db.collection.update( { field: value }, { $push: { field: value1 } } );
```

Here, `$push` appends `value1` to the array identified by `value` in `field`. Be aware of the following behaviors:

– If the field specified in the `$push` statement (e.g. `{ $push: { field: value1 } }`) does not exist in the matched document, the operation adds a new array with the specified field and value (e.g. `value1`) to the matched document.

– The operation will fail if the field specified in the `$push` statement is *not* an array. `$push` does not fail when pushing a value to a non-existent field.

– If `value1` is an array itself, `$push` appends the whole array as an element in the identified array. To add multiple items to an array, use `$pushAll`.

$pushAll

$pushAll

The $pushAll operator is similar to the $push but adds the ability to append several values to an array at once.

```
db.collection.update( { field: value }, { $pushAll: { field1: [ value1, value2, value3 ] } } )
```

Here, $pushAll appends the values in [value1, value2, value3] to the array in field1 in the document matched by the statement { field: value } in collection.

If you specify a single value, $pushAll will behave as $push.

$query

$query

The $query (page 460) operator provides an interface to describe queries. Consider the following operation.

```
db.collection.find()._addSpecial( "$query" : { value : 100 } )
```

This is equivalent to the following db.collection.find() method that may be more familiar to you:

```
db.collection.find( { value : 100 } )
```

$regex

$regex

The $regex operator provides regular expression capabilities in queries. MongoDB uses Perl compatible regular expressions (i.e. "PCRE."))The following examples are equivalent:

```
db.collection.find( { field: /acme.*corp/i } );
db.collection.find( { field: { $regex: 'acme.*corp', $options: 'i' } } );
```

These expressions match all documents in collection where the value of field matches the case-insensitive regular expression acme.*corp.

$regex uses "Perl Compatible Regular Expressions" (PCRE) as the matching engine.

$options

$regex provides four option flags:

- i toggles case insensitivity, and allows all letters in the pattern to match upper and lower cases.
- m toggles multiline regular expression. Without this option, all regular expression match within one line.

  If there are no newline characters (e.g. \n) or no start/end of line construct, the m option has no effect.
- x toggles an "extended" capability. When set, $regex ignores all white space characters unless escaped or included in a character class.

  Additionally, it ignores characters between an un-escaped # character and the next new line, so that you may include comments in complicated patterns. This only applies to data characters; white space characters may never appear within special character sequences in a pattern.

  The x option does not affect the handling of the VT character (i.e. code 11.)

New in version 1.9.0.

- s allows the dot (e.g. .) character to match all characters *including* newline characters.

`$regex` only provides the `i` and `m` options in the short JavaScript syntax (i.e. `http://docs.mongodb.org/manual/acme.*corp/i`). To use `x` and `s` you must use the "`$regex`" operator with the "`$options`" syntax.

To combine a regular expression match with other operators, you need to specify the "`$regex`" operator. For example:

```
db.collection.find( { field: $regex: /acme.*corp/i, $nin: [ 'acmeblahcorp' ] } );
```

This expression returns all instances of `field` in `collection` that match the case insensitive regular expression `acme.*corp` that *don't* match `acmeblahcorp`.

`$regex` uses *indexes* only when the regular expression has an anchor for the beginning (i.e. `^`) of a string. Additionally, while `http://docs.mongodb.org/manual/^a/`, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` are equivalent, they have different performance characteristics. All of these expressions use an index if an appropriate index exists; however, `http://docs.mongodb.org/manual/^a.*/`, and `http://docs.mongodb.org/manual/^a.*$/` are slower. `http://docs.mongodb.org/manual/^a/` can stop scanning after matching the prefix.

$rename

`$rename`

New in version 1.7.2. *Syntax*: `{$rename: { <old name1>: <new name1>, <old name2>: <new name2>, ... } }`

The `$rename` operator updates the name of a field. The new field name must differ from the existing field name.

Consider the following example:

```
db.students.update( { _id: 1 }, { $rename: { 'nickname': 'alias', 'cell': 'mobile' } } )
```

This operation renames the field `nickname` to `alias`, and the field `cell` to `mobile`.

If the document already has a field with the *new* field name, the `$rename` operator removes that field and renames the field with the *old* field name to the *new* field name.

The `$rename` operator will expand arrays and sub-documents to find a match for field names. When renaming a field in a sub-document to another sub-document or to a regular field, the sub-document itself remains.

Consider the following examples involving the sub-document of the following document:

```
{ "_id": 1,
  "alias": [ "The American Cincinnatus", "The American Fabius" ],
  "mobile": "555-555-5555",
  "nmae": { "first" : "george", "last" : "washington" }
}
```

–To rename a sub-document, call the `$rename` operator with the name of the sub-document as you would any other field:

```
db.students.update( { _id: 1 }, { $rename: { "nmae": "name" } } )
```

This operation renames the sub-document `nmae` to `name`:

```
{ "_id": 1,
  "alias": [ "The American Cincinnatus", "The American Fabius" ],
  "mobile": "555-555-5555",
  "name": { "first" : "george", "last" : "washington" }
}
```

– To rename a field within a sub-document, call the `$rename` operator using the dot notation to refer to the field. Include the name of the sub-document in the new field name to ensure the field remains in the sub-document:

```
db.students.update( { _id: 1 }, { $rename: { "name.first": "name.fname" } } )
```

This operation renames the sub-document field `first` to `fname`:

```
{ "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "mobile" : "555-555-5555",
  "name" : { "fname" : "george", "last" : "washington" }
}
```

– To rename a field within a sub-document and move it to another sub-document, call the `$rename` operator using the dot notation to refer to the field. Include the name of the new sub-document in the `new name`:

```
db.students.update( { _id: 1 }, { $rename: { "name.last": "contact.lname" } } )
```

This operation renames the sub-document field `last` to `lname` and moves it to the sub-document `contact`:

```
{ "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "contact" : { "lname" : "washington" },
  "mobile" : "555-555-5555",
  "name" : { "fname" : "george" }
}
```

If the new field name does not include a sub-document name, the field moves out of the subdocument and becomes a regular document field.

Consider the following behavior when the specified old field name does not exist:

– When renaming a single field and the existing field name refers to a non-existing field, the `$rename` operator does nothing, as in the following:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'spouse' } } )
```

This operation does nothing because there is no field named `wife`.

– When renaming multiple fields and **all** of the old field names refer to non-existing fields, the `$rename` operator does nothing, as in the following:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'spouse',
                                             'vice': 'vp',
                                             'office': 'term' } } )
```

This operation does nothing because there are no fields named `wife`, `vice`, and `office`.

–When renaming multiple fields and **some** but not all old field names refer to non-existing fields, the `$rename` operator performs the following operations: Changed in version 2.2.

∗Renames the fields that exist to the specified new field names.

∗Ignores the non-existing fields.

Consider the following query that renames both an existing field `mobile` and a non-existing field `wife`. The field named `wife` does not exist and `$rename` sets the field to a name that already exists `alias`.

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'alias',
                                             'mobile': 'cell' } } )
```

This operation renames the `mobile` field to `cell`, and has no other impact action occurs.

```
{ "_id" : 1,
  "alias" : [ "The American Cincinnatus", "The American Fabius" ],
  "cell" : "555-555-5555",
  "name" : { "lname" : "washington" },
  "places" : { "d" : "Mt Vernon", "b" : "Colonial Beach" }
}
```

Note: Before version 2.2, when renaming multiple fields and only some (but not all) old field names refer to non-existing fields:

∗For the fields with the old names that do exist, the `$rename` operator renames these fields to the specified new field names.

∗For the fields with the old names that do **not** exist:

·**if** no field exists with the new field name, the `$rename` operator does nothing.

·**if** fields already exist with the new field names, the `$rename` operator drops these fields.

Consider the following operation that renames both the field `mobile`, which exists, and the field `wife`, which does not exist. The operation tries to set the field named `wife` to `alias`, which is the name of an existing field:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'alias', 'mobile': 'cell' } } )
```

Before 2.2, the operation renames the field `mobile` to `cell` *and* drops the `alias` field even though the field `wife` does not exist:

```
{ "_id" : 1,
  "cell" : "555-555-5555",
  "name" : { "lname" : "washington" },
  "places" : { "d" : "Mt Vernon", "b" : "Colonial Beach" }
}
```

$returnKey

`$returnKey`

Only return the index key (i.e. *_id*) or keys for the results of the query. Use the following form:

```
db.collection.find()._addSpecial("$returnKey" , true )
```

$set

$set

Use the `$set` operator to set a particular value. The `$set` operator requires the following syntax:

```
db.collection.update( { field: value1 }, { $set: { field1: value2 } } );
```

This statement updates in the document in `collection` where `field` matches `value1` by replacing the value of the field `field1` with `value2`. This operator will add the specified field or fields if they do not exist in this document *or* replace the existing value of the specified field(s) if they already exist.

$showDiskLoc

$showDiskLoc

Use the following modifier to display the disk location:

```
db.collection.find()._addSpecial("$showDiskLoc" , true)
```

$size

$size

The `$size` operator matches any array with the number of elements specified by the argument. For example:

```
db.collection.find( { field: { $size: 2 } } );
```

returns all documents in `collection` where `field` is an array with 2 or more elements. For instance, the above expression will return `{ field: [ red, green ] }` and `{ field: [ apple, lime ] }` but *not* `{ field: fruit }` or `{ field: [ orange, lemon, grapefruit ] }`. To match fields with only one element within an array use `$size` with a value of 1, as follows:

```
db.collection.find( { field: { $size: 1 } } );
```

`$size` does not accept ranges of values. To select documents based on fields with different numbers of elements, create a counter field that you increment when you add elements to a field.

Queries cannot use indexes for the `$size` portion of a query, although the other portions of a query can use indexes if applicable.

$snapshot

$snapshot

The `$snapshot` (page 464) operator ensures that the results returned by a query:

–contains no duplicates.

–misses no objects.

–returns all matching objects that were present at the beginning and the end of the query.

Snapshot mode does not guarantee the inclusion (or omission) of an object present at the beginning of the query but not at the end (due to an update.) Use the following syntax:

```
db.foo.find()._addSpecial( "$snapshot", true )
```

The JavaScript function `cursor.snapshot()` provides equivalent functionality in the `mongo` shell. See the following example, which is equivalent to the above:

```
db.foo.find().snapshot()
```

Do not use snapshot with $hint (page 449), or $orderby (page 457) (cursor.sort().)

$type

$type

Syntax: { field: { $type: <BSON type> } }

$type selects the documents where the *value* of the field is the specified *BSON* type.

Consider the following example:

```
db.inventory.find( { price: { $type : 1 } } )
```

This query will select all documents in the inventory collection where the price field value is a Double.

If the field holds an array, the $type operator performs the type check against the array elements and **not** the field.

Consider the following example where the tags field holds an array:

```
db.inventory.find( { tags: { $type : 4 } } )
```

This query will select all documents in the inventory collection where the tags array contains an element that is itself an array.

If instead you want to determine whether the tags field is an array type, use the $where operator:

```
db.inventory.find( { $where : "Array.isArray(this.tags)" } )
```

See the SERVER-1475 for more information about the array type.

Refer to the following table for the available *BSON* types and their corresponding numbers.

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

MinKey and MaxKey compare less than and greater than all other possible *BSON* element values, respectively, and exist primarily for internal use.

Note: To query if a field value is a `MinKey`, you must use the `$type` with `-1` as in the following example:

```
db.collection.find( { field: { $type: -1 } } )
```

Example

Consider the following example operation sequence that demonstrates both type comparison *and* the special `MinKey` and `MaxKey` values:

```
db.test.insert( {x : 3});
db.test.insert( {x : 2.9} );
db.test.insert( {x : new Date()} );
db.test.insert( {x : true } );
db.test.insert( {x : MaxKey } )
db.test.insert( {x : MinKey } )

db.test.find().sort({x:1})
{ "_id" : ObjectId("4b04094b7c65b846e2090112"), "x" : { $minKey : 1 } }
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Jul 25 2012 18:42:03 GMT-0500 (ES
{ "_id" : ObjectId("4b0409487c65b846e2090111"), "x" : { $maxKey : 1 } }
```

To query for the minimum value of a *shard key* of a *sharded cluster*, use the following operation when connected to the `mongos` (page 676):

```
use config
db.chunks.find( { "min.shardKey": { $type: -1 } } )
```

Warning: Storing values of the different types in the same field in a collection is *strongly* discouraged.

See Also:

`find()`, `insert()`, `$where`, *BSON*, *shard key*, *sharded cluster* .

$uniqueDocs

`$uniqueDocs`

New in version 2.0. For *geospatial* queries, MongoDB may return a single document more than once for a single query, because geospatial indexes may include multiple coordinate pairs in a single document, and therefore return the same document more than once.

The `$uniqueDocs` operator inverts the default behavior of the `$within` operator. By default, the `$within` operator returns the document only once. If you specify a value of `false` for `$uniqueDocs`, MongoDB will return multiple instances of a single document.

Example

Given an `addressBook` collection with a document in the following form:

```
{ addresses: [ { name: "Home", loc: [55.5, 42.3] }, { name: "Work", loc: [32.3, 44.2] } ] }
```

The following query would return the same document multiple times:

```
db.addressBook.find( { "addresses.loc": { "$within": { "$box": [ [0,0], [100,100] ], $unique
```

The following query would return each matching document, only once:

```
db.addressBook.find( { "address.loc": { "$within": { "$box": [ [0,0], [100,100] ], $uniqueDo
```

You cannot specify `$uniqueDocs` with `$near` or haystack queries.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$unset

$unset

The `$unset` operator deletes a particular field. Consider the following example:

```
db.collection.update( { field: value1 }, { $unset: { field1: "" } } );
```

The above example deletes `field1` in `collection` from documents where `field` has a value of `value1`. The value of specified for the value of the field in the `$unset` statement (i.e. `""` above,) does not impact the operation.

If documents match the initial query (e.g. `{ field: value1 }` above) but do not have the field specified in the `$unset` operation, (e.g. `field1`) there the statement has no effect on the document.

$where

$where

Use the `$where` operator to pass a string containing a JavaScript expression to the query system to provide greater flexibility with queries. Consider the following:

```
db.collection.find( { $where: "this.a == this.b" } );
```

Warning: `$where` evaluates JavaScript and cannot take advantage of indexes. Therefore, query performance improves when you express your query using the standard MongoDB operators (e.g., `$gt`, `$in`).
In general, you should use `$where` only when you can't express your query using another operator. If you must use `$where`, try to include at least one other standard query operator to filter the result set. Using `$where` alone requires a table scan.

$within

$within

The `$within` operator allows you to select items that exist within a shape on a coordinate system for *geospatial* queries. This operator uses the following syntax:

```
db.collection.find( { location: { $within: { shape } } } );
```

Replace `{ shape }` with a document that describes a shape. The `$within` command supports three shapes. These shapes and the relevant expressions follow:

–Rectangles. Use the `$box` operator, consider the following variable and `$within` document:

```
db.collection.find( { location: { $within: { $box: [[100,0], [120,100]] } } } );
```

Here a box, `[[100,120], [100,0]]` describes the parameter for the query. As a minimum, you must specify the lower-left and upper-right corners of the box.

–Circles. Use the `$center` operator. Specify circles in the following form:

```
db.collection.find( { location: { $within: { $center: [ center, radius ] } } } );
```

–Polygons. Use the `$polygon` operator. Specify polygons with an array of points. See the following example:

```
db.collection.find( { location: { $within: { $box: [[100,120], [100,100], [120,100], [24
```

The last point of a polygon is implicitly connected to the first point.

All shapes include the border of the shape as part of the shape, although this is subject to the imprecision of floating point numbers.

Use `$uniqueDocs` to control whether documents with many location fields show up multiple times when more than one of its fields match the query.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

`$box`

New in version 1.4. The `$box` operator specifies a rectangular shape for the `$within` operator in *geospatial* queries. To use the `$box` operator, you must specify the bottom left and top right corners of the rectangle in an array object. Consider the following example:

```
db.collection.find( { loc: { $within: { $box: [ [0,0], [100,100] ] } } } )
```

This will return all the documents that are within the box having points at: `[0,0]`, `[0,100]`, `[100,0]`, and `[100,100]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

`$polygon`

New in version 1.9. Use `$polygon` to specify a polygon for a bounded query using the `$within` operator for *geospatial* queries. To define the polygon, you must specify an array of coordinate points, as in the following:

[[x1,y1], [x2,y2], [x3,y3]]

The last point specified is always implicitly connected to the first. You can specify as many points, and therefore sides, as you like. Consider the following bounded query for documents with coordinates within a polygon:

```
db.collection.find( { loc: { $within: { $polygon: [ [0,0], [3,6], [6,0]  ] } } } )
```

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$center

New in version 1.4. This specifies a circle shape for the `$within` operator in *geospatial* queries. To define the bounds of a query using `$center`, you must specify:

–the center point, and

–the radius

Considering the following example:

```
db.collection.find( { location: { $within: { $center: [ [0,0], 10 ] } } } );
```

The above command returns all the documents that fall within a 10 unit radius of the point `[0,0]`.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

$uniqueDocs

New in version 2.0. For *geospatial* queries, MongoDB may return a single document more than once for a single query, because geospatial indexes may include multiple coordinate pairs in a single document, and therefore return the same document more than once.

The `$uniqueDocs` operator inverts the default behavior of the `$within` operator. By default, the `$within` operator returns the document only once. If you specify a value of `false` for `$uniqueDocs`, MongoDB will return multiple instances of a single document.

Example

Given an `addressBook` collection with a document in the following form:

```
{ addresses: [ { name: "Home", loc: [55.5, 42.3] }, { name: "Work", loc: [32.3, 44.2] }
```

The following query would return the same document multiple times:

```
db.addressBook.find( { "addresses.loc": { "$within": { "$box": [ [0,0], [100,100] ], $un
```

The following query would return each matching document, only once:

```
db.addressBook.find( { "address.loc": { "$within": { "$box": [ [0,0], [100,100] ], $uniq
```

You cannot specify `$uniqueDocs` with `$near` or haystack queries.

Note: A geospatial index *must* exist on a field holding coordinates before using any of the geolocation query operators.

- Projection operators:

$elemMatch (projection)

See Also:

$elemMatch (query) (page 447)

$elemMatch

New in version 2.2. Use the $elemMatch (page 469) projection operator to limit the response of a query to a single matching element of an array. Consider the following:

Example

Given the following document fragment:

```
{
 _id: ObjectId(),
 zipcode: 63109,
 dependents: [
              { name: "john", school: 102, age: 10 },
              { name: "jess", school: 102, age: 11 },
              { name: "jeff", school: 108, age: 15 }
             ]
}
```

Consider the following `find()` operation:

```
var projection = { _id: 0, dependents: { $elemMatch: { school: 102 }}};
db.students.find( { zipcode: 63109 }, projection);
```

The query would return all documents where the value of the `zipcode` field is `63109`, while the projection excludes the `_id` field and only includes the first matching element of the `dependents` array where the `school` element has a value of `102`. The documents would take the following form:

```
{
 dependents: [
              { name: "john", school: 102, age: 10 }
             ]
}
```

Note: The $elemMatch (page 469) projection will only match one array element per source document.

$slice (projection)

$slice

The $slice (page 470) operator controls the number of items of an array that a query returns. Consider the following prototype query:

```
db.collection.find( { field: value }, { array: {$slice: count } } );
```

This operation selects the document `collection` identified by a field named `field` that holds `value` and returns the number of elements specified by the value of `count` from the array stored in the `array` field. If `count` has a value greater than the number of elements in `array` the query returns all elements of the array.

$slice (page 470) accepts arguments in a number of formats, including negative values and arrays. Consider the following examples:

```
db.posts.find( {}, { comments: { $slice: 5 } } )
```

Here, $slice (page 470) selects the first five items in an array in the `comments` field.

```
db.posts.find( {}, { comments: { $slice: -5 } } )
```

This operation returns the last five items in array.

The following examples specify an array as an argument to slice. Arrays take the form of `[ skip , limit ]`, where the first value indicates the number of items in the array to skip and the second value indicates the number of items to return.

```
db.posts.find( {}, { comments: { $slice: [ 20, 10 ] } } )
```

Here, the query will only return 10 items, after skipping the first 20 items of that array.

```
db.posts.find( {}, { comments: { $slice: [ -20, 10 ] } } )
```

This operation returns 10 items as well, beginning with the item that is 20th from the last item of the array.

- Aggregation operators:

$add (aggregation)

$add
Takes an array of one or more numbers and adds them together, returning the sum.

$addToSet (aggregation)

$addToSet
Returns an array of all the values found in the selected field among the documents in that group. *Every unique value only appears once* in the result set. There is no ordering guarantee for the output documents.

$and (aggregation)

$and
Takes an array one or more values and returns `true` if *all* of the values in the array are `true`. Otherwise `$and` returns `false`.

Note: `$and` uses short-circuit logic: the operation stops evaluation after encountering the first `false` expression.

$avg (aggregation)

$avg
Returns the average of all the values of the field in all documents selected by this group.

$cmp (aggregation)

$cmp
Takes two values in an array and returns an integer. The returned value is:

–A negative number if the first value is less than the second.

–A positive number if the first value is greater than the second.

–`0` if the two values are equal.

$cond (aggregation)

$cond

Use the $cond (page 472) operator with the following syntax:

```
{ $cond: [ <boolean-expression>, <true-case>, <false-case> ] }
```

Takes an array with three expressions, where the first expression evaluates to a Boolean value. If the first expression evaluates to true, $cond (page 472) returns the value of the second expression. If the first expression evaluates to false, $cond (page 472) evaluates and returns the third expression.

$dayOfMonth (aggregation)

$dayOfMonth

Takes a date and returns the day of the month as a number between 1 and 31.

$dayOfWeek (aggregation)

$dayOfWeek

Takes a date and returns the day of the week as a number between 1 (Sunday) and 7 (Saturday.)

$dayOfYear (aggregation)

$dayOfYear

Takes a date and returns the day of the year as a number between 1 and 366.

$divide (aggregation)

$divide

Takes an array that contains a pair of numbers and returns the value of the first number divided by the second number.

$eq (aggregation)

$eq

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

$first (aggregation)

$first

Returns the first value it encounters for its group .

Note: Only use $first (page 473) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$group (aggregation)

$group

Groups documents together for the purpose of calculating aggregate values based on a collection of documents. Practically, group often supports tasks such as average page views for each page in a website on a daily basis.

The output of $group (page 473) depends on how you define groups. Begin by specifying an identifier (i.e. a _id field) for the group you're creating with this pipeline. You can specify a single field from the documents in the pipeline, a previously computed value, or an aggregate key made up from several incoming fields. Aggregate keys may resemble the following document:

```
{ _id : { author: '$author', pageViews: '$pageViews', posted: '$posted' } }
```

With the exception of the _id field, $group (page 473) cannot output nested documents.

Every group expression must specify an _id field. You may specify the _id field as a dotted field path reference, a document with multiple fields enclosed in braces (i.e. { and }), or a constant value.

Note: Use $project (page 477) as needed to rename the grouped field after an $group (page 473) operation, if necessary.

Consider the following example:

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

This groups by the author field and computes two fields, the first docsPerAuthor is a counter field that adds one for each document with a given author field using the $sum (page 481) function. The viewsPerAuthor field is the sum of all of the pageViews fields in the documents for each group.

Each field defined for the $group (page 473) must use one of the group aggregation function listed below to generate its composite value:

$addToSet

Returns an array of all the values found in the selected field among the documents in that group. *Every unique value only appears once* in the result set. There is no ordering guarantee for the output documents.

$first

Returns the first value it encounters for its group .

Note: Only use $first (page 473) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$last

Returns the last value it encounters for its group.

Note: Only use $last (page 475) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$max
Returns the highest value among all values of the field in all documents selected by this group.

$min
Returns the lowest value among all values of the field in all documents selected by this group.

$avg
Returns the average of all the values of the field in all documents selected by this group.

$push
Returns an array of all the values found in the selected field among the documents in that group. *A value may appear more than once* in the result set if more than one field in the grouped documents has that value.

$sum
Returns the sum of all the values for a specified field in the grouped documents, as in the second use above.

Alternately, if you specify a value as an argument, `$sum` (page 481) will increment this field by the specified value for every document in the grouping. Typically, as in the first use above, specify a value of `1` " in order to count members of the group.

Warning: The aggregation system currently stores `$group` (page 473) operations in memory, which may cause problems when processing a larger number of groups.

$gt (aggregation)

$gt
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than* the second value.
- `false` when the first value is *less than or equal to* the second value.

$gte (aggregation)

$gte
Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *greater than or equal* to the second value.
- `false` when the first value is *less than* the second value.

$hour (aggregation)

$hour
Takes a date and returns the hour between 0 and 23.

$ifNull (aggregation)

$ifNull
Use the `$ifNull` (page 474) operator with the following syntax:

```
{ $ifNull: [ <expression>, <replacement-if-null> ] }
```

Takes an array with two expressions. $ifNull (page 474) returns the first expression if it evaluates to a non-null value. Otherwise, $ifNull (page 474) returns the second expression's value.

$last (aggregation)

$last

Returns the last value it encounters for its group.

Note: Only use $last (page 475) when the $group (page 473) follows an $sort (page 479) operation. Otherwise, the result of this operation is unpredictable.

$limit (aggregation)

$limit

Restricts the number of *documents* that pass through the $limit (page 475) in the *pipeline*.

$limit (page 475) takes a single numeric (positive whole number) value as a parameter. Once the specified number of documents pass through the pipeline operator, no more will. Consider the following example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. $limit (page 475) has no effect on the content of the documents it passes.

$lt (aggregation)

$lt

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than* the second value.
- `false` when the first value is *greater than or equal to* the second value.

$lte (aggregation)

$lte

Takes two values in an array and returns an integer. The returned value is:

- `true` when the first value is *less than or equal to* the second value.
- `false` when the first value is *greater than* the second value.

$match (aggregation)

$match

Provides a query-like interface to filter documents out of the aggregation *pipeline*. The $match (page 475) drops documents that do not match the condition from the aggregation pipeline, and it passes documents that match along the pipeline unaltered.

The syntax passed to the $match (page 475) is identical to the *query* syntax. Consider the following prototype form:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

The following example performs a simple field equality test:

```
db.article.aggregate(
    { $match : { author : "dave" } }
);
```

This operation only returns documents where the author field holds the value dave. Consider the following example, which performs a range test:

```
db.article.aggregate(
    { $match : { score  : { $gt : 50, $lte : 90 } } }
);
```

Here, all documents return when the score field holds a value that is greater than 50 and less than or equal to 90.

Note: Place the $match (page 475) as early in the aggregation *pipeline* as possible. Because $match (page 475) limits the total number of documents in the aggregation pipeline, earlier $match (page 475) operations minimize the amount of later processing. If you place a $match (page 475) at the very beginning of a pipeline, the query can take advantage of *indexes* like any other db.collection.find() or db.collection.findOne().

Warning: You cannot use $where or *geospatial operations* in $match (page 475) queries as part of the aggregation pipeline.

$max (aggregation)

$max

Returns the highest value among all values of the field in all documents selected by this group.

$min (aggregation)

$min

Returns the lowest value among all values of the field in all documents selected by this group.

$minute (aggregation)

$minute

Takes a date and returns the minute between 0 and 59.

$mod (aggregation)

$mod

Takes an array that contains a pair of numbers and returns the *remainder* of the first number divided by the second number.

See Also:

`$mod`

$month (aggregation)

`$month`

Takes a date and returns the month as a number between 1 and 12.

$multiply (aggregation)

`$multiply`

Takes an array of one or more numbers and multiples them, returning the resulting product.

$ne (aggregation)

`$ne`

Takes two values in an array returns an integer. The returned value is:

- `true` when the values are **not equivalent**.
- `false` when the values are **equivalent**.

$not (aggregation)

`$not`

Returns the boolean opposite value passed to it. When passed a `true` value, `$not` returns `false`; when passed a `false` value, `$not` returns `true`.

$or (aggregation)

`$or`

Takes an array of one or more values and returns `true` if *any* of the values in the array are `true`. Otherwise `$or` returns false.

Note: `$or` uses short-circuit logic: the operation stops evaluation after encountering the first `true` expression.

$project (aggregation)

`$project`

Reshapes a document stream by renaming, adding, or removing fields. Also use `$project` (page 477) to create computed values or sub-objects. Use `$project` (page 477) to:

- Include fields from the original document.
- Insert computed fields.
- Rename fields.
- Create and populate fields that hold sub-documents.

Use $project (page 477) to quickly select the fields that you want to include or exclude from the response. Consider the following aggregation framework operation.

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
 );
```

This operation includes the title field and the author field in the document that returns from the aggregation *pipeline*.

Note: The _id field is always included by default. You may explicitly exclude _id as follows:

```
db.article.aggregate(
    { $project : {
        _id : 0 ,
        title : 1 ,
        author : 1
    }}
);
```

Here, the projection excludes the _id field but includes the title and author fields.

Projections can also add computed fields to the document stream passing through the pipeline. A computed field can use any of the *expression operators* (page 276). Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1,
        doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

Here, the field doctoredPageViews represents the value of the pageViews field after adding 10 to the original field using the $add (page 471).

Note: You must enclose the expression that defines the computed field in braces, so that the expression is a valid object.

You may also use $project (page 477) to rename fields. Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        page_views : "$pageViews" ,
        bar : "$other.foo"
    }}
);
```

This operation renames the pageViews field to page_views, and renames the foo field in the other sub-document as the top-level field bar. The field references used for renaming fields are direct expressions and do not use an operator or surrounding braces. All aggregation field references can use dotted paths to refer to fields in nested documents.

Finally, you can use the $project (page 477) to create and populate new sub-documents. Consider the following example that creates a new object-valued field named stats that holds a number of values:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {
            pv : "$pageViews",
            foo : "$other.foo",
            dpv : { $add:["$pageViews", 10] }
        }
    }}
);
```

This projection includes the `title` field and places `$project` (page 477) into "inclusive" mode. Then, it creates the `stats` documents with the following fields:

- `pv` which includes and renames the `pageViews` from the top level of the original documents.
- `foo` which includes the value of `other.foo` from the original documents.
- `dpv` which is a computed field that adds 10 to the value of the `pageViews` field in the original document using the `$add` (page 471) aggregation expression.

$push (aggregation)

$push

Returns an array of all the values found in the selected field among the documents in that group. *A value may appear more than once* in the result set if more than one field in the grouped documents has that value.

$second (aggregation)

$second

Takes a date and returns the second between 0 and 59, but can be 60 to account for leap seconds.

$skip (aggregation)

$skip

Skips over the specified number of *documents* that pass through the `$skip` (page 479) in the *pipeline* before passing all of the remaining input.

`$skip` (page 479) takes a single numeric (positive whole number) value as a parameter. Once the operation has skipped the specified number of documents, it passes all the remaining documents along the *pipeline* without alteration. Consider the following example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This operation skips the first 5 documents passed to it by the pipeline. `$skip` (page 479) has no effect on the content of the documents it passes along the pipeline.

$sort (aggregation)

$sort

The `$sort` (page 479) *pipeline* operator sorts all input documents and returns them to the pipeline in sorted order. Consider the following prototype form:

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

This sorts the documents in the collection named `<collection-name>`, according to the key and specification in the `{ <sort-key> }` document.

Specify the sort in a document with a field or fields that you want to sort by and a value of `1` or `-1` to specify an ascending or descending sort respectively, as in the following example:

```
db.users.aggregate(
    { $sort : { age : -1, posts: 1 } }
);
```

This operation sorts the documents in the `users` collection, in descending order according by the `age` field and then in ascending order according to the value in the `posts` field.

Note: The `$sort` (page 479) cannot begin sorting documents until previous operators in the pipeline have returned all output.

–`$skip` (page 479)

`$sort` (page 479) operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators:

–`$project` (page 477)

–`$unwind` (page 481)

–`$group` (page 473).

Warning: Unless the `$sort` (page 479) operator can use an index, in the current release, the sort must fit within memory. This may cause problems when sorting large numbers of documents.

$strcasecmp (aggregation)

`$strcasecmp`

Takes in two strings. Returns a number. `$strcasecmp` (page 480) is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. `$strcasecmp` (page 480) returns 0 if the strings are identical.

Note: `$strcasecmp` (page 480) may not make sense when applied to glyphs outside the Roman alphabet.

`$strcasecmp` (page 480) internally capitalizes strings before comparing them to provide a case-*insensitive* comparison. Use `$cmp` (page 471) for a case sensitive comparison.

$substr (aggregation)

`$substr`

`$substr` (page 480) takes a string and two numbers. The first number represents the number of bytes in the string to skip, and the second number specifies the number of bytes to return from the string.

Note: $substr (page 480) is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$subtract (aggregation)

$subtract

Takes an array that contains a pair of numbers and subtracts the second from the first, returning their difference.

$sum (aggregation)

$sum

Returns the sum of all the values for a specified field in the grouped documents, as in the second use above.

Alternately, if you specify a value as an argument, $sum (page 481) will increment this field by the specified value for every document in the grouping. Typically, as in the first use above, specify a value of 1 " in order to count members of the group.

$toLower (aggregation)

$toLower

Takes a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase.

Note: $toLower (page 481) may not make sense when applied to glyphs outside the Roman alphabet.

$toUpper (aggregation)

$toUpper

Takes a single string and converts that string to uppercase, returning the result. All lowercase letters become uppercase.

Note: $toUpper (page 481) may not make sense when applied to glyphs outside the Roman alphabet.

$unwind (aggregation)

$unwind

Peels off the elements of an array individually, and returns a stream of documents. $unwind (page 481) returns one document for every member of the unwound array within every source document. Take the following aggregation command:

```
db.article.aggregate(
    { $project : {
        author : 1 ,
        title : 1 ,
        tags : 1
    }},
    { $unwind : "$tags" }
);
```

Note: The dollar sign (i.e. `$`) must proceed the field specification handed to the `$unwind` (page 481) operator.

In the above aggregation `$project` (page 477) selects (inclusively) the `author`, `title`, and `tags` fields, as well as the `_id` field implicitly. Then the pipeline passes the results of the projection to the `$unwind` (page 481) operator, which will unwind the `tags` field. This operation may return a sequence of documents that resemble the following for a collection that contains one document holding a `tags` field with an array of 3 items.

```
{
     "result" : [
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "good"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             }
     ],
     "OK" : 1
}
```

A single document becomes 3 documents: each document is identical except for the value of the `tags` field. Each value of `tags` is one of the values in the original "tags" array.

Note: `$unwind` (page 481) has the following behaviors:

- `$unwind` (page 481) is most useful in combination with `$group` (page 473).
- You may undo the effects of unwind operation with the `$group` (page 473) pipeline operator.
- If you specify a target field for `$unwind` (page 481) that does not exist in an input document, the pipeline ignores the input document, and will generate no result documents.
- If you specify a target field for `$unwind` (page 481) that is not an array, `aggregate()` generates an error.
- If you specify a target field for `$unwind` (page 481) that holds an empty array (`[]`) in an input document, the pipeline ignores the input document, and will generates no result documents.

$week (aggregation)

`$week`

Takes a date and returns the week of the year as a number between 0 and 53.

Weeks begin on Sundays, and week 1 begins with the first Sunday of the year. Days preceding the first Sunday of the year are in week 0. This behavior is the same as the "%U" operator to the strftime standard library function.

$year (aggregation)

$year
Takes a date and returns the full year.

36.1.2 Database Commands

addShard

addShard

Options

- **name** – Optional. Unless specified, a name will be automatically provided to uniquely identify the shard.
- **maxSize** – Optional. Unless specified, shards will consume the total amount of available space on their machines if necessary. Use the maxSize value to limit the amount of space the database can use. Specify this value in megabytes.

The addShard command registers a new with a sharded cluster. You must run this command against a mongos (page 676) instance. The command takes the following form:

```
{ addShard: "<hostname>:<port>" }
```

Replace <hostname>:<port> with the hostname and port of the database instance you want to add as a shard. Because the mongos (page 676) instances do not have state and distribute configuration in the *config database*, send this command to only one mongos (page 676) instance.

Note: 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 (page 642) exceeds the value of maxSize.

---

aggregate

aggregate
New in version 2.1.0. aggregate implements the *aggregation framework*. Consider the following prototype form:

```
{ aggregate: "[collection]", pipeline: [pipeline] }
```

Where [collection] specifies the name of the collection that contains the data that you wish to aggregate. The pipeline argument holds an array that contains the specification for the aggregation operation. Consider the following example from the *aggregation documentation* (page 253).

```
db.runCommand(
{ aggregate : "article", pipeline : [
  { $project : {
     author : 1,
```

```
      tags : 1,
   } },
   { $unwind : "$tags" },
   { $group : {
      _id : "$tags",
      authors : { $addToSet : "$author" }
   } }
 ] }
);
```

More typically this operation would use the `aggregate` helper in the `mongo` shell, and would resemble the following:

```
db.article.aggregate(
   { $project : {
      author : 1,
      tags : 1,
   } },
   { $unwind : "$tags" },
   { $group : {
      _id : "$tags",
      authors : { $addToSet : "$author" }
   } }
);
```

For more aggregation documentation, please see:

- *Aggregation Framework* (page 253)
- *Aggregation Framework Reference* (page 269)
- *Aggregation Framework Examples* (page 259)

applyOps (internal)

applyOps

Parameters

- **operations** (*array*) – an array of operations to perform.
- **preCondition** (*array*) – Optional. Defines one or more conditions that the destination must meet applying the entries from the `<operations>` array. `ns` to specify a *namespace*, `q` to specify a *query* and `res` to specify the result that the query should match. You may specify zero, one, or many `preCondition` documents.

`applyOps` provides a way to apply entries from an *oplog* created by *replica set* members and *master* instances in a master/*slave* deployment. `applyOps` is primarily an internal command to support sharding functionality, and has the following prototype form:

```
db.runCommand( { applyOps: [ <operations> ], preCondition: [ { ns: <namespace>, q: <query>, res:
```

`applyOps` applies oplog entries from the `<operations>` array, to the `mongod` instance. The `preCondition` array provides the ability to specify conditions that must be true in order to apply the oplog entry.

You can specify as many `preCondition` sets as needed. If you specify the `ns` option, `applyOps` will only apply oplog entries for the *collection* described by that namespace. You may also specify a query in the `q` field with a corresponding expected result in the `res` field that must match in order to apply the oplog entry.

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

authenticate

authenticate

Clients use `authenticate` to authenticate a connection. When using the shell, use the command helper as follows:

```
db.authenticate( "username", "password" )
```

availableQueryOptions (internal)

availableQueryOptions

`availableQueryOptions` is an internal command that is only available on `mongos` (page 676) instances.

buildInfo

buildInfo

The `buildInfo` command is an administrative command which returns a build summary for the current `mongod`.

```
{ buildInfo: 1 }
```

```
The information provided includes the following:
```

- The version of MongoDB currently running.
- The information about the system that built the "`mongod`" binary, including a timestamp for the build.
- The architecture of the binary (i.e. 64 or 32 bits.)
- The maximum allowable *BSON* object size in bytes (in the field `maxBsonObjectSize`.)

checkShardingIndex (internal)

checkShardingIndex

`checkShardingIndex` is an internal command that supports the sharding functionality.

clean (internal)

clean

`clean` is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

clone

clone

The clone command clone a database from a remote MongoDB instance to the current host. clone copies the database on the remote instance with the same name as the current database. The command takes the following form:

```
{ clone: "db1.example.net:27017" }
```

Replace db1.example.net:27017 above with the resolvable hostname for the MongoDB instance you wish to copy from. Note the following behaviors:

- clone can run against a *slave* or a non-*primary* member of a *replica set*.
- clone does not snapshot the database. If any clients update the database you're copying at any point during the clone operation, the resulting database may be inconsistent.
- You must run clone on the **destination server**.
- The destination server is not locked for the duration of the clone operation. This means that clone will occasionally yield to allow other operations to complete.

See copydb for similar functionality.

Warning: This command obtains an intermittent write-lock on the destination server, that can block other operations until it completes.

cloneCollection

cloneCollection

The cloneCollection command copies a collection from a remote server to the server where you run the command.

Parameters

- **from** – Specify a resolvable hostname, and optional port number of the remote server where the specified collection resides.
- **query** – Optional. A query document, in the form of a *document*, that filters the documents in the remote collection that cloneCollection will copy to the current database. See db.collection.find().
- **copyIndexes** (*Boolean*) – Optional. true by default. When set to false the indexes on the originating server are *not* copied with the documents in the collection.

Consider the following example:

```
{ cloneCollection: "users", from: "db.example.net:27017", query: { active: true }, copyIndexes:
```

This operation copies the "users" collection from the current database on the server at db.example.net. The operation only copies documents that satisfy the query { active: true } and does not copy indexes. cloneCollection copies indexes by default, but you can disable this behavior by setting { copyIndexes: false }. The query and copyIndexes arguments are optional.

cloneCollection creates a collection on the current database with the same name as the origin collection. If, in the above example, the users collection already exists, then MongoDB appends documents in the remote collection to the destination collection.

cloneCollectionAsCapped

cloneCollectionAsCapped

The `cloneCollectionAsCapped` command creates a new *capped collection* from an existing, non-capped collection within the same database. The operation does not affect the original non-capped collection.

The command has the following syntax:

```
{ cloneCollectionAsCapped: <existing collection>, toCollection: <capped collection>, size: <capp
```

The command copies an `existing collection` and creates a new `capped collection` with a maximum size specified by the `capped size` in bytes. The name of the new capped collection must be distinct and cannot be the same as that of the original existing collection. To replace the original non-capped collection with a capped collection, use the `convertToCapped` command.

During the cloning, the `cloneCollectionAsCapped` command exhibit the following behavior:

- MongoDB will transverse the documents in the original collection in *natural order* as they're loaded.
- If the `capped size` specified for the new collection collection is smaller than the size of the original uncapped collection, then MongoDB will begin overwriting earlier documents in insertion order, which is *first in, first out* (e.g "FIFO").

closeAllDatabases (internal)

closeAllDatabases

`closeAllDatabases` is an internal command that invalidates all cursors and closes the open database files. The next operation that uses the database will reopen the file.

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

collMod

collMod

New in version 2.2. `collMod` makes it possible to add flags to a collection to modify the behavior of MongoDB. In the current release the only available flag is `usePowerOf2Sizes`. The command takes the following prototype form:

```
db.runCommand( {"collMod" : [collection] , "[flag]" : [value]" } )
```

In this command substitute `[collection]` with the name of the collection, and `[flag]` and `[value]` with the flag and value you want to set.

usePowerOf2Sizes

The `usePowerOf2Sizes` flag changes the method that MongoDB uses to allocate space on disk for documents in this collection. By setting `usePowerOf2Sizes`, you ensure that MongoDB will allocate space for documents in sizes that are powers of 2 (e.g. 4, 8, 16, 32, 64, 128, 256, 512...8388608). With this option MongoDB will be able to more effectively reuse space.

`usePowerOf2Sizes` is useful for collections where you will be inserting and deleting large numbers of documents to ensure that MongoDB will effectively use space on disk.

Note: Changed in version 2.2.1: If you're using `usePowerOf2Sizes`, ensure you use at least 2.2.1 to avoid the issue described in SERVER-7238. `usePowerOf2Sizes` only affects subsequent allocations cased by document insertion or record relocation as a result of document growth, and *does not* affect existing allocations.

collStats

collStats

The `collStats` command returns a variety of storage statistics for a given collection. Use the following syntax:

```
{ collStats: "database.collection" , scale : 1024 }
```

Specify a namespace `database.collection` and use the `scale` argument to scale the output. The above example will display values in kilobytes.

Examine the following example output, which uses the `db.collection.stats()` helper in the `mongo` shell.

```
> db.users.stats()
{
        "ns" : "app.users",              // namespace
        "count" : 9,                     // number of documents
        "size" : 432,                    // collection size in bytes
        "avgObjSize" : 48,               // average object size in bytes
        "storageSize" : 3840,            // (pre)allocated space for the collection
        "numExtents" : 1,                // number of extents (contiguously allocated chunks of d
        "nindexes" : 2,                  // number of indexes
        "lastExtentSize" : 3840,         // size of the most recently created extent
        "paddingFactor" : 1,             // padding can speed up updates if documents grow
        "flags" : 1,
        "totalIndexSize" : 16384,        // total index size in bytes
        "indexSizes" : {                 // size of specific indexes in bytes
                "_id_" : 8192,
                "username" : 8192
        },
        "ok" : 1
}
```

Note: The scale factor rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

See Also:

"*Collection Statistics Reference* (page 653)."

compact

compact

New in version 2.0. The `compact` command rewrites and defragments a single collection. Additionally, the command drops all indexes at the beginning of compaction and rebuilds the indexes at the end. `compact` is conceptually similar to `repairDatabase`, but works on a single collection rather than an entire database.

The command has the following syntax:

```
{ compact: <collection name> }
```

You may also specify the following options:

Parameters

- **force** – Changed in version 2.2: `compact` blocks activities only for the database it is compacting. The `force` specifies whether the `compact` command can run on the primary node in a *replica set*. Set to `true` to run the `compact` command on the primary node in a *replica set*. Otherwise, the `compact` command returns an error when invoked on a *replica set* primary because the command blocks all other activity.

- **paddingFactor** – New in version v2.2. *Default:* `1.0`

  Minimum: `1.0` (no padding.)

  Maximum: `4.0`

  The `paddingFactor` describes the *record size* allocated for each document as a factor of the document size. If your updates increase the size of the documents, padding will increase the amount of space allocated to each document and avoid expensive document relocation operations within the data files.

  You can calculate the padding size by subtracting the document size from the record size or, in terms of the `paddingFactor`, by subtracting `1` from the `paddingFactor`:

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

  For example, a `paddingFactor` of `1.0` specifies a padding size of `0` whereas a `paddingFactor` of `1.2` specifies a padding size of `0.2` or 20 percent (20%) of the document size.

  With the following command, you can use the `paddingFactor` option of the `compact` command to set the record size to `1.1` of the document size, or a padding factor of 10 percent (10%):

  ```
  db.runCommand ( { compact: '<collection>', paddingFactor: 1.1 } )
  ```

- **paddingBytes** – New in version 2.2. The `paddingBytes` sets the padding as an absolute number of bytes. Specifying `paddingBytes` can be useful if your documents start small but then increase in size significantly. For example,if your documents are initially 40 bytes long and you grow them by 1KB, using `paddingBytes: 1024` might be reasonable since using `paddingFactor: 4.0` would specify a record size of 160 bytes (`4.0` times the initial document size), which would only provide a padding of 120 bytes (i.e. record size of 160 bytes minus the document size).

  With the following command, you can use the `paddingBytes` option of the `compact` command to set the padding size to `100` bytes on the collection named by `<collection>`:

  ```
  db.runCommand ( { compact: '<collection>', paddingBytes: 100 } )
  ```

Warning: Always have an up-to-date backup before performing server maintenance such as the `compact` operation.

Note the following behaviors:

- `compact` blocks all other activity. In MongoDB 2.2, `compact` blocks activities only for its database. You may view the intermediate progress either by viewing the the `mongod` log file, or by running the `db.currentOp()` in another shell instance.

- `compact` removes any *padding factor* in the collection when issued without either the `paddingFactor` option or the `paddingBytes` option. This may impact performance if the documents grow regularly. However, `compact` retains existing paddingFactor statistics for the collection that MongoDB will use to calculate the padding factor for future inserts.

- `compact` generally uses less disk space than `repairDatabase` and is faster. However,the `compact` command is still slow and does block other database use. Only use `compact` during scheduled maintenance periods.
- If you terminate the operation with the `db.killOp()` method or restart the server before it has finished:
  - If you have journaling enabled, the data remains consistent and usable, regardless of the state of the `compact` operation. You may have to manually rebuild the indexes.
  - If you do not have journaling enabled and the `mongod` or `compact` terminates during the operation, it's impossible to guarantee that the data is in a consistent state.
  - In either case, much of the existing free space in the collection may become un-reusable. In this scenario, you should rerun the compaction to completion to restore the use of this free space.
- `compact` may increase the total size and number of our data files, especially when run for the first time. However, this will not increase the total collection storage space since storage size is the amount of data allocated within the database files, and not the size/number of the files on the file system.
- `compact` requires a small amount of additional disk space while running but unlike `repairDatabase` it does *not* free space on the file system.
- You may also wish to run the `collStats` command before and after compaction to see how the storage space changes for the collection.
- `compact` commands do not replicate to secondaries in a *replica set*:
  - Compact each member separately.
  - Ideally, compaction runs on a secondary. See option `force:true` above for information regarding compacting the primary.
  - If you run `compact` on a secondary, the secondary will enter a "recovering" state to prevent clients from sending read operations during compaction. Once the compaction finishes the secondary will automatically return to secondary state.

    You may refer to the "partial script for automating step down and compaction") for an example.
- `compact` is a command issued to a `mongod`. In a sharded environment, run `compact` on each shard separately as a maintenance operation.
- It is not possible to compact *capped collections* because they don't have padding, and documents cannot grow in these collections. However, the documents of a *capped collections* are not subject to fragmentation.

See Also:

`repairDatabase`

connPoolStats

connPoolStats

Note: `connPoolStats` only returns meaningful results for `mongos` (page 676) instances and for `mongod` instances in sharded clusters.

The command `connPoolStats` returns information regarding the number of open connections to the current database instance, including client connections and server-to-server connections for replication and clustering. The command takes the following form:

```
{ connPoolStats: 1 }
```

The value of the argument (i.e. `1`) does not affect the output of the command. See *Connection Pool Statistics Reference* (page 657) for full documentation of the `connPoolStats` output.

connPoolSync (internal)

connPoolSync

`connPoolSync` is an internal command.

convertToCapped

convertToCapped

The `convertToCapped` command converts an existing, non-capped collection to a *capped collection* within the same database.

The command has the following syntax:

```
{convertToCapped: <collection>, size: <capped size> }
```

`convertToCapped` takes an existing collection (`<collection>`) and transforms it into a capped collection with a maximum size in bytes, specified to the `size` argument (`<capped size>`).

During the conversion process, the `convertToCapped` command exhibit the following behavior:

- MongoDB transverses the documents in the original collection in *natural order* and loads the documents into into a new capped collection.
- If the `capped size` specified for the capped collection is smaller than the size of the original uncapped collection, then MongoDB will overwrite documents from of the collection based on insertion order, or *first in, first out* order.
- Internally, to convert the collection, MongoDB uses the following procedure
  - `cloneCollectionAsCapped` command creates the capped collection and imports the data.
  - MongoDB drops the original collection.
  - `renameCollection` renames the new capped collection to the name of the original collection.

Note: MongoDB does not support the `convertToCapped` command in a sharded cluster.

Warning: The `convertToCapped` will not recreate indexes from the original collection on the new collection. If you need indexes on this collection you will need to create these indexes after the conversion is complete.

See Also:

`create`

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

copydb

copydb

The `copydb` command copies a database from a remote host to the current host. The command has the following syntax:

```
{ copydb: 1:
  fromhost: <hostname>,
  fromdb: <db>,
  todb: <db>,
  slaveOk: <bool>,
  username: <username>,
  password: <password>,
  nonce: <nonce>,
  key: <key> }
```

All of the following arguments are optional:

- slaveOK
- username
- password
- nonce
- key

You can omit the `fromhost` argument, to copy one database to another database within a single MongoDB instance.

You must run this command on the the destination, or the `todb` server.

Be aware of the following behaviors:

- `copydb` can run against a *slave* or a non-*primary* member of a *replica set*. In this case, you must set the `slaveOk` option to `true`.
- `copydb` does not snapshot the database. If the state of the database changes at any point during the operation, the resulting database may be inconsistent.
- You must run `copydb` on the **destination server**.
- The destination server is not locked for the duration of the `copydb` operation. This means that `copydb` will occasionally yield to allow other operations to complete.
- If the remote server has authentication enabled, then you must include a username and password. You must also include a nonce and a key. The nonce is a one-time password that you request from the remote server using the `copydbgetnonce` command. The `key` is a hash generated as follows:

  ```
  hex_md5(nonce + username + hex_md5(username + ":mongo:" + pass))
  ```

  If you need to copy a database and authenticate, it's easiest to use the shell helper:

  ```
  db.copyDatabase(<remove_db_name>, <local_db_name>, <from_host_name>, <username>, <password>)
  ```

copydbgetnonce (internal)

copydbgetnonce

Client libraries use `copydbgetnonce` to get a one-time password for use with the `copydb` command.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed; however, the write lock for this operation is short lived.

---

count

count

The `count` command counts the number of documents in a collection. For example:

```
> db.runCommand( { count: "collection" } );
{ "n" : 10 , "ok" : 1 }
```

In the `mongo` shell, this returns the number of documents in the collection (e.g. `collection`.) You may also use the `count()` method on any cursor object to return a count of the number of documents in that cursor. The following operation produces the same result in the `mongo` shell:

```
> db.collection.count():
{ "n" : 10 , "ok" : 1 }
```

The collection in this example has 10 documents.

create

create

The `create` command explicitly creates a collection. The command uses the following syntax:

```
{ create: <collection_name> }
```

To create a *capped collection* limited to 40 KB, issue command in the following form:

```
{ create: "collection", capped: true, size: 40 * 1024 }
```

The options for creating capped collections are:

Options

- **capped** – Specify `true` to create a *capped collection*.
- **autoIndexId** – Specify `false` to disable the automatic index created on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 698) for more information.
- **size** – The maximum size for the capped collection. Once a capped collection reaches its max size, MongoDB will drop old documents from the database to make way for the new documents. You must specify a `size` argument for all capped collections.
- **max** – The maximum number of documents to preserve in the capped collection. This limit is subject to the overall size of the capped collection. If a capped collection reaches its maximum size before it contains the maximum number of documents, the database will remove old documents. Thus, if you use this option, ensure that the total size for the capped collection is sufficient to contain the max.

The `db.createCollection()` (page 552) provides a wrapper function that provides access to this functionality.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed. The write lock for this operation is typically short lived; however, allocations for large capped collections may take longer.

cursorInfo

cursorInfo

The `cursorInfo` command returns information about current cursor allotment and use. Use the following form:

```
{ cursorInfo: 1 }
```

The value (e.g. `1` above,) does not effect the output of the command.

`cursorInfo` returns the total number of open cursors (`totalOpen`,) the size of client cursors in current use (`clientCursors_size`,) and the number of timed out cursors since the last server restart (`timedOut`.)

dataSize

dataSize

For internal use.

The `dataSize` command returns the size data size for a set of data within a certain range:

```
{ dataSize: "database.collection", keyPattern: { field: 1 }, min: { field: 10 }, max: { field: 1
```

This will return a document that contains the size of all matching documents. Replace `database.collection` value with database and collection from your deployment. The `keyPattern`, `min`, and `max` parameters are options.

The amount of time required to return `dataSize` depends on the amount of data in the collection.

dbHash (internal)

dbHash

`dbHash` is an internal command.

dbStats

dbStats

The `dbStats` command returns storage statistics for a given database. The command takes the following syntax:

```
{ dbStats: 1, scale: 1 }
```

The value of the argument (e.g. `1` above) to `dbStats` does not affect the output of the command. The `scale` option allows you to specify how to scale byte values. For example, a `scale` value of `1024` will display the results in kilobytes rather than in bytes.

The time required to run the command depends on the total size of the database. Because the command has to touch all data files, the command may take several seconds to run.

In the `mongo` shell, the `db.stats()` function provides a wrapper around this functionality. See the "*Database Statistics Reference* (page 651)" document for an overview of this output.

diagLogging (internal)

diagLogging

`diagLogging` is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

distinct

distinct

The `distinct` command returns an array of distinct values for a given field across a single collection. The command takes the following form:

```
{ distinct: collection, key: age, query: { field: { $exists: true } } }
```

This operation returns all distinct values of the field (or `key`) `age` in documents that match the query `{ field: { $exists: true }`.

Note: The query portion of the `distinct` is optional.

The shell and many *drivers* provide a helper method that provides this functionality. You may prefer the following equivalent syntax:

```
db.collection.distinct("age", { field: { $exists: true } } );
```

The `distinct` command will use an index to locate and return data.

driverOIDTest (internal)

driverOIDTest

`driverOIDTest` is an internal command.

drop

drop

The `drop` command removes an entire collection from a database. The command has following syntax:

```
{ drop: <collection_name> }
```

The `mongo` shell provides the equivalent helper method:

```
db.collection.drop();
```

Note that this command also removes any indexes associated with the dropped collection.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

dropDatabase

dropDatabase

The `dropDatabase` command drops a database, deleting the associated data files. `dropDatabase` operates on the current database.

In the shell issue the `use <database>` command, replacing `<database>` with the name of the database you wish to delete. Then use the following command form:

```
{ dropDatabase: 1 }
```

The `mongo` shell also provides the following equivalent helper method:

```
db.dropDatabase();
```

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

dropIndexes

dropIndexes

The `dropIndexes` command drops one or all indexes from the current collection. To drop all indexes, issue the command like so:

```
{ dropIndexes: "collection", index: "*" }
```

To drop a single, issue the command by specifying the name of the index you want to drop. For example, to drop the index named `age_1`, use the following command:

```
{ dropIndexes: "collection", index: "age_1" }
```

The shell provides a useful command helper. Here's the equivalent command:

```
db.collection.dropIndex("age_1");
```

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

emptycapped

emptycapped

The `emptycapped` command removes all documents from a capped collection. Use the following syntax:

```
{ emptycapped: "events" }
```

This command removes all records from the capped collection named `events`.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

enableSharding

enableSharding

The `enableSharding` command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: 1 }
```

Once you've enabled sharding in a database, you can use the `shardCollection` command to begin the process of distributing data among the shards.

eval

`eval`

The `eval` command evaluates JavaScript functions on the database server. Consider the following (trivial) example:

```
{ eval: function() { return 3+3 } }
```

The shell also provides a helper method, so you can express the above as follows:

```
db.eval( function { return 3+3 } } );
```

The shell's JavaScript interpreter evaluates functions entered directly into the shell. If you want to use the server's interpreter, you must run `eval`.

Be aware of following behaviors and limitations:

- `eval` does not work in *sharded* environments.
- The `eval` operation take a write lock by default. This means that writes to database aren't permitted while it's running. You can, however, disable the lock by setting the `nolock` flag to `true`. For example:

```
{ eval: function() { return 3+3 }, nolock: true }
```

> **Warning:** Do not disable the write lock if the operation may modify the contents of the database in anyway.

There are some circumstances where the `eval()` implements a strictly-read only operation that need not block other operations when disabling the write lock may be useful. Use this functionality with extreme caution.

features (internal)

`features`

`features` is an internal command that returns the build-level feature settings.

filemd5

`filemd5`

The `filemd5` command returns the *md5* hashes for a single files stored using the *GridFS* specification. Client libraries use this command to verify that files are correctly written to MongoDB. The command takes the `files_id` of the file in question and the name of the GridFS root collection as arguments. For example:

```
{ filemd5: ObjectId("4f1f10e37671b50e4ecd2776"), root: "fs" }
```

findAndModify

`findAndModify`

The `findAndModify` command atomically modifies and returns a single document. By default, the returned

document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option.

The command has the following syntax:

```
{ findAndModify: <collection>,  <options>  }
```

The `findAndModify` command takes the following are sub-document `options`:

Fields

- **query** (*document*) – Optional. Specifies the selection criteria for the modification. The `query` field employs the same *query selectors* as used in the `db.collection.find()` method. Although the query may match multiple documents, `findAndModify` will only select one document to modify.

  The `query` field has the following syntax:

  ```
  query: { <query expression> }
  ```

- **sort** (*document*) – Optional. Determines which document the operation will modify if the query selects multiple documents. `findAndModify` will modify the first document in the sort order specified by this argument.

  The `sort` field has the following syntax:

  ```
  sort: { field1: value1, field2: value2, ... }
  ```

- **remove** (*boolean*) – Optional if `update` field exists. When `true`, removes the selected document. The default is `false`.

  The `remove` field has the following syntax:

  ```
  remove: <boolean>
  ```

- **update** (*document*) – Optional if `remove` field exists. Performs an update of the selected document. The `update` field employs the same *update operators* or `field: value` specifications to modify the selected document.

  ```
  update: { <update expression> }
  ```

- **new** (*boolean*) – Optional. When `true`, returns the modified document rather than the original. The `findAndModify` method ignores the `new` option for `remove` operations. The default is `false`.

  ```
  new: <boolean>
  ```

- **fields** (*document*) – Optional. A subset of fields to return.

  ```
  fields: { field1: <boolean>, field2: <boolean> ... }
  ```

- **upsert** (*boolean*) – Optional. Used in conjunction with the `update` field. When `true`, the `findAndModify` command creates a new document if the `query` returns no documents. The default is `false`. In version 2.2, the `findAndModify` command returns `null` when `upsert` is `true`.

  ```
  upsert: <boolean>
  ```

Changed in version 2.2: Previously, `upsert` operations returned a an empty document (e.g. `{ }`,) see *the 2.2 release notes* (page 696) for more information.

Consider the following example:

```
{ findAndModify: "people",
  query: { name: "Tom", state: "active", rating: { $gt: 10 } },
  sort: { rating: 1 },
  update: { $inc: { score: 1 } }
}
```

This command performs the following actions:

1. The `query` finds a document in the `people` collection where the `name` field has the value `Tom`, the `state` field has the value `active` and the `rating` field has a value `greater than` 10.
2. The `sort` orders the results of the query in ascending order.
3. The update `increments` the value of the `score` field by 1.
4. The command returns the original unmodified document selected for this update.

The shell and many *drivers* provide a `findAndModify()` helper method. Using the shell helper, this same operation can take the following form:

```
db.people.findAndModify( {
    query: { name: "Tom", state: "active", rating: { $gt: 10 } },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } }
    } );
```

Warning: When using `findAndModify` in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` operations issued against `mongos` (page 676) instances for non-sharded collections function normally.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed; however, typically the write lock is short lived and equivalent to other similar `update()` operations.

---

flushRouterConfig

`flushRouterConfig`

`flushRouterConfig` clears the current cluster information cached by a `mongos` (page 676) instance and reloads all *sharded cluster* metadata from the *config database*.

This forces an update when the configuration database holds data that is newer that the data cached in the `mongos` (page 676) process.

Warning: Do not modify the config data, except as explicitly documented. A config database cannot typically tolerate manual manipulation.

`flushRouterConfig` is an administrative command that is only available for `mongos` (page 676) instances. New in version 1.8.2.

forceerror (internal)

forceerror

The `forceerror` command is for testing purposes only. Use `forceerror` to force a user assertion exception. This command always returns an `ok` value of 0.

fsync

fsync

The `fsync` command forces the `mongod` process to flush all pending writes to the storage layer. `mongod` is always writing data to the storage layer as applications write more data to the database. MongoDB guarantees that it will write all data to disk within the `syncdelay` (page 627) interval, which is 60 seconds by default.

```
{ fsync: 1 }
```

The `fsync` operation is synchronous by default, to run `fsync` asynchronously, use the following form:

```
{ fsync: 1, async: true }
```

The connection will return immediately. You can check the output of `db.currentOp()` for the status of the `fsync` operation.

The primary use of `fsync` is to lock the database during backup operations. This will flush all data to the data storage layer and block all write operations until you unlock the database. Consider the following command form:

```
{ fsync: 1, lock: true }
```

Note: You may continue to perform read operations on a database that has a `fsync` lock. However, following the first write operation all subsequent read operations wait until you unlock the database.

To check on the current state of the fsync lock, use `db.currentOp()`. Use the following JavaScript function in the shell to test if the database is currently locked:

```
serverIsLocked = function () {
                     var co = db.currentOp();
                     if (co && co.fsyncLock) {
                         return true;
                     }
                     return false;
                 }
```

After loading this function into your `mongo` shell session you can call it as follows:

```
serverIsLocked()
```

This function will return `true` if the database is currently locked and `false` if the database is not locked. To unlock the database, make a request for an unlock using the following command:

```
db.getSiblingDB("admin").$cmd.sys.unlock.findOne();
```

New in version 1.9.0: The `db.fsyncLock()` and `db.fsyncUnlock()` helpers in the shell. In the `mongo` shell, you may use the `db.fsyncLock()` and `db.fsyncUnlock()` wrappers for the `fsync` lock and unlock process:

```
db.fsyncLock();
db.fsyncUnlock();
```

Note: `fsync` lock is only possible on individual shards of a sharded cluster, not on the entire sharded cluster. To backup an entire sharded cluster, please read *considerations for backing up sharded clusters* (page 186).

If your `mongod` has *journaling* enabled, consider using *another method* (page 181) to back up your database.

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)
```

geoNear

geoNear

The `geoNear` command provides an alternative to the `$near` operator. In addition to the functionality of `$near`, `geoNear` returns the distance of each item from the specified point along with additional diagnostic information. For example:

```
{ geoNear : "places" , near : [50,50], num : 10 }
```

Here, `geoNear` returns the 10 items nearest to the coordinates `[50,50]` in the collection named `places`. *geoNear* provides the following options (specify all distances in the same units as the document coordinate system:)

Fields

- **near** – Takes the coordinates (e.g. `[ x, y ]`) to use as the center of a geospatial query.
- **num** – Specifies the maximum number of documents to return.
- **maxDistance** – Optional. Limits the results to those falling within a given distance of the center coordinate.
- **query** – Optional. Further narrows the results using any standard MongoDB query operator or selection. See `db.collection.find()` and "`http://docs.mongodb.org/manual/reference/operators`" for more information.
- **spherical** – Optional. Default: `false`. When `true` MongoDB will return the query as if the coordinate system references points on a spherical plane rather than a plane.
- **distanceMultiplier** – Optional. Specifies a factor to multiply all distances returned by `geoNear`. For example, use `distanceMultiplier` to convert from spherical queries returned in radians to linear units (i.e. miles or kilometers) by multiplying by the radius of the Earth.
- **includeLocs** – Optional. Default: `false`. When specified `true`, the query will return the location of the matching documents in the result.
- **uniqueDocs** – Optional. Default `true`. The default settings will only return a matching document once, even if more than one of its location fields match the query. When `false` the query will return documents with multiple matching location fields more than once. See `$uniqueDocs` for more information on this option

geoSearch

geoSearch

The `geoSearch` command provides an interface to MongoDB's *haystack index* functionality. These indexes are useful for returning results based on location coordinates *after* collecting results based on some other query (i.e. a "haystack.") Consider the following example:

```
{ geoSearch : "places", near : [33, 33], maxDistance : 6, search : { type : "restaurant" }, limi
```

The above command returns all documents with a `type` of `restaurant` having a maximum distance of 6 units from the coordinates `[30,33]` in the collection `places` up to a maximum of 30 results.

Unless specified otherwise, the `geoSearch` command limits results to 50 documents.

geoWalk

geoWalk

`geoWalk` is an internal command.

getCmdLineOpts

getCmdLineOpts

The `getCmdLineOpts` command returns a document containing command line options used to start the given `mongod`:

```
{ getCmdLineOpts: 1 }
```

This command returns a document with two fields, `argv` and `parsed`. The `argv` field contains an array with each item from the command string used to invoke `mongod`. The document in the `parsed` field includes all runtime options, including those parsed from the command line and those specified in the configuration file, if specified.

Consider the following example output of `getCmdLineOpts`:

```
{
        "argv" : [
                "/usr/bin/mongod",
                "--config",
                "/etc/mongodb.conf",
                "--fork"
        ],
        "parsed" : {
                "bind_ip" : "127.0.0.1",
                "config" : "/etc/mongodb/mongodb.conf",
                "dbpath" : "/srv/mongodb",
                "fork" : true,
                "logappend" : "true",
                "logpath" : "/var/log/mongodb/mongod.log",
                "quiet" : "true"
        },
        "ok" : 1
}
```

http://docs.mongodb.org/manual/administration/import-export/

getLastError

getLastError

The `getLastError` command returns the error status of the last operation on the *current connection*. By default MongoDB does not provide a response to confirm the success or failure of a write operation, clients typically use `getLastError` in combination with write operations to ensure that the write succeeds.

Consider the following prototype form.

```
{ getLastError: 1 }
```

The following options are available:

Parameters

- **j** (*boolean*) – If `true`, wait for the next journal commit before returning, rather than a full disk flush. If `mongod` does not have journaling enabled, this option has no effect.
- **w** – When running with replication, this is the number of servers to replica to before returning. A `w` value of 1 indicates the primary only. A `w` value of 2 includes the primary and at least one secondary, etc. In place of a number, you may also set `w` to `majority` to indicate that the command should wait until the latest write propagates to a majority of replica set members. If using `w`, you should also use `wtimeout`. Specifying a value for `w` without also providing a `wtimeout` may cause `getLastError` to block indefinitely.
- **fsync** (*boolean*) – If `true`, wait for `mongod` to write this data to disk before returning. Defaults to false. In most cases, use the `j` option to ensure durability and consistency of the data set.
- **wtimeout** (*integer*) – Optional. Milliseconds. Specify a value in milliseconds to control how long the to wait for write propagation to complete. If replication does not complete in the given timeframe, the `getLastError` command will return with an error status.

See Also:

"*Replica Set Write Concern* (page 54)" and "`db.getLastError()`."

getLog

getLog

The `getLog` command returns a document with a `log` array that contains recent messages from the `mongod` process log. The `getLog` command has the following syntax:

```
{ getLog: <log> }
```

Replace `<log>` with one of the following values:

- `global` - returns the combined output of all recent log entries.
- `rs` - if the `mongod` is part of a *replica set*, `getLog` will return recent notices related to replica set activity.
- `startupWarnings` - will return logs that *may* contain errors or warnings from MongoDB's log from when the current process started. If `mongod` started without warnings, this filter may return an empty array.

You may also specify an asterisk (e.g. `*`) as the `<log>` value to return a list of available log filters. The following interaction from the `mongo` shell connected to a replica set:

```
db.adminCommand({getLog: "*" })
{ "names" : [ "global", "rs", "startupWarnings" ], "ok" : 1 }
```

getLog returns events from a RAM cache of the mongod events and *does not* read log data from the log :file.

getParameter

getParameter

getParameter is an administrative command for retrieving the value of options normally set on the command line. Issue commands against the *admin database* as follows:

```
{ getParameter: 1, <option>: 1 }
```

The values specified for getParameter and <option> do not affect the output. The command works with the following options:

- **quiet**
- **notablescan**
- **logLevel**
- **syncdelay**

See Also:

setParameter for more about these parameters.

getPrevError

getPrevError

The getPrevError command returns the errors since the last resetError command.

See Also:

db.getPrevError()

getShardMap (internal)

getShardMap

getShardMap is an internal command that supports the sharding functionality.

getShardVersion (internal)

getShardVersion

getShardVersion is an internal command that supports sharding functionality.

getnonce (internal)

getnonce

Client libraries use getnonce to generate a one-time password for authentication.

getoptime (internal)

getoptime

getoptime is an internal command.

godinsert (internal)

godinsert

`godinsert` is an internal command for testing purposes only.

Note: This command obtains a write lock on the affected database and will block other operations until it has completed.

---

group

group

The `group` command returns an array of grouped items. `group` provides functionality analogous to the `GROUP BY` statement in SQL. Consider the following example:

```
db.users.runCommand( { group:
                       { key: { school_id: true },
                         cond: { active: 1 },
                         reduce: function(obj, prev) { obj.total += 1; },
                         initial: { total: 0 }
                        } } );
```

More typically, in the `mongo` shell, you will call the `group` command using the `group()` method. Consider the following form:

```
db.users.group( { key: { school_id: true },
                  cond: { active: 1 },
                  reduce: function(obj, prev) { obj.total += 1; },
                  initial: { total: 0 }
                } );
```

In these examples `group` runs against the collection `users` and counts the total number of active users from each school. Fields allowed by the group command include:

Fields

- **key** (*document*) – Specify one or more fields to group by. Use the form of a *document*.
- **reduce** – Specify a reduce function that operates over all the iterated objects. Typically these aggregator functions perform some sort of summing or counting. The reduce function takes two arguments: the current document and an aggregation counter object.
- **initial** – The starting value of the aggregation counter object.
- **keyf** – Optional. A function that returns a "key object" for use as the grouping key. Use `keyf` instead of `key` to specify a key that is not a single/multiple existing fields. For example, use `keyf` to group by day or week in place of a fixed `key`.
- **cond** – Optional. A statement that must evaluate to true for the `db.collection.group()` to process this document. Essentially this argument specifies a query document (as for `db.collection.find()`). Unless specified, `db.collection.group()` runs the "reduce" function against all documents in the collection.
- **finalize** – Optional. A function that runs each item in the result set before `db.collection.group()` returns the final value. This function can either modify the document by computing and adding an average field, or return compute and return a new document.

Note: The result set of the `db.collection.group()` must fit within the size *maximum BSON document* (page 679). Furthermore, you must ensure that there are fewer then 10,000 unique keys. If you have more than this, use `mapReduce`.

Warning: `group()` does not work in *shard environments*. Use the *aggregation framework* or *map-reduce* (i.e. **mapReduce** in *sharded environments*.

handshake (internal)

handshake

`handshake` is an internal command.

isMaster

isMaster

The `isMaster` command provides a basic overview of the current replication configuration. MongoDB *drivers* and *clients* use this command to determine what kind of member they're connected to and to discover additional members of a *replica set*. The `db.isMaster()` method provides a wrapper around this database command.

The command takes the following form:

```
{ isMaster: 1 }
```

This command returns a *document* containing the following fields:

`isMaster.`**setname**
The name of the current replica set, if applicable.

`isMaster.`**ismaster**
A boolean value that reports when this node is writable. If `true`, then the current node is either a *primary* node in a *replica set*, a *master* node in a master-slave configuration, of a standalone `mongod`.

`isMaster.`**secondary**
A boolean value that, when `true`, indicates that the current node is a *secondary* member of a *replica set*.

`isMaster.`**hosts**
An array of strings in the format of "`[hostname]:[port]`" listing all nodes in the *replica set* that are not "*hidden*".

`isMaster.`**primary**
The `[hostname]:[port]` for the current *replica set primary*, if applicable.

`isMaster.`**me**
The `[hostname]:[port]` of the node responding to this command.

`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.`**localTime**
New in version 2.1.1. Returns the local server time in UTC. This value is a *ISOdate*. You can use the `toString()` JavaScript method to convert this value to a local date string, as in the following example:

```
db.isMaster().localTime.toString();
```

isSelf (internal)

_isSelf

_isSelf is an internal command.

isdbgrid

isdbgrid

This command verifies that a process is a mongos (page 676).

If you issue the isdbgrid command when connected to a mongos (page 676), the response document includes the isdbgrid field set to 1. The returned document is similar to the following:

```
{ "isdbgrid" : 1, "hostname" : "app.example.net", "ok" : 1 }
```

If you issue the isdbgrid command when connected to a mongod, MongoDB returns an error document. The isdbgrid command is not available to mongod. The error document, however, also includes a line that reads "isdbgrid" : 1, just as in the document returned for a mongos (page 676). The error document is similar to the following:

```
{
   "errmsg" : "no such cmd: isdbgrid",
   "bad cmd" : {
         "isdbgrid" : 1
   },
   "ok" : 0
}
```

You can instead use the isMaster command to determine connection to a mongos (page 676). When connected to a mongos (page 676), the isMaster command returns a document that contains the string isdbgrid in the msg field.

journalLatencyTest

journalLatencyTest

journalLatencyTest (page 507) is an administrative command that tests the length of time required to write and perform a file system sync (e.g. *fsync*) for a file in the journal directory. You must issue the journalLatencyTest (page 507) command against the *admin database* in the form:

```
{ journalLatencyTest: 1 }
```

The value (i.e. 1 above), does not affect the operation of the command.

listCommands

listCommands

The listCommands command generates a list of all database commands implemented for the current mongod instance.

listDatabases

listDatabases

The listDatabases command provides a list of existing databases along with basic statistics about them:

```
{ listDatabases: 1 }
```

The value (e.g. 1) does not effect the output of the command. listDatabases returns a document for each database Each document contains a name field with the database name, a sizeOnDisk field with the total size of the database file on disk in bytes, and an empty field specifying whether the database has any data.

listShards

listShards

Use the listShards command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

logRotate

logRotate

The logRotate command is an administrative command that allows you to rotate the MongoDB logs to prevent a single logfile from consuming too much disk space. You must issue the logRotate command against the *admin database* in the form:

```
{ logRotate: 1 }
```

Note: Your mongod instance needs to be running with the *--logpath [file]* (page 582) option.

You may also rotate the logs by sending a SIGUSR1 signal to the mongod process. If your mongod has a process ID of 2200, here's how to send the signal on Linux:

```
kill -SIGUSR1 2200
```

logRotate renames the existing log file by appending the current timestamp to the filename. The appended timestamp has the following form:

```
<YYYY>-<mm>-<DD>T<HH>-<MM>-<SS>
```

Then logRotate creates a new log file with the same name as originally specified by the logpath (page 622) setting to mongod or mongos (page 676).

Note: New in version 2.0.3: The logRotate command is available to mongod instances running on Windows systems with MongoDB release 2.0.3 and higher.

---

logout

logout

The logout command terminates the current authenticated session:

```
{ logout: 1 }
```

Note: If you're not logged in and using authentication, this command will have no effect.

---

mapReduce

mapReduce

The `mapReduce` command allows you to run map-reduce-style aggregations over a collection.

Options

- **map** – A JavaScript function that performs the "map" step of the map-reduce operation. This function references the current input document and calls the `emit(key,value)` method that supplies values to the reduce function. Map functions may call `emit()`, once, more than once, or not at all depending on the type of aggregation.
- **reduce** – A JavaScript function that performs the "reduce" step of the MapReduce operation. The reduce function receives an array of emitted values from the map function, and returns a single value. Because it's possible to invoke the reduce function more than once for the same key, the structure of the object returned by function must be identical to the structure of the emitted function.
- **out** – Specifies the location of the out of the reduce stage of the operation. Specify a string to write the output of the Map/Reduce job to a collection with that name. The map-reduce operation will replace the content of the specified collection in the current database by default. See below for additional options.
- **query** – Optional. A query object, like the query used by the `db.collection.find()` method. Use this to filter to limit the number of documents enter the map phase of the aggregation.
- **sort** – Optional. Sorts the input objects using this key. This option is useful for optimizing the job. Common uses include sorting by the emit key so that there are fewer reduces.
- **limit** – Optional. Species a maximum number of objects to return from the collection.
- **finalize** – Optional. Specifies an optional "finalize" function to run on a result, following the reduce stage, to modify or control the output of the `mapReduce` operation.
- **scope** – Optional. Place a *document* as the contents of this field, to place fields into the global javascript scope.
- **jsMode** (*Boolean*) – Optional. The `jsMode` option defaults to `false`.
- **verbose** (*Boolean*) – Optional. The `verbose` option provides statistics on job execution times.

`mapReduce` only require `map` and `reduce` options, all other fields are optional. You must write all `map` and `reduce` functions in JavaScript.

The `out` field of the `mapReduce`, provides a number of additional configuration options that you may use to control how MongoDB returns data from the map-reduce job. Consider the following 4 output possibilities.

Parameters

- **replace** – Optional. Specify a collection name (e.g. `{ out: { replace: collectionName } }`) where the output of the map-reduce overwrites the contents of the collection specified (i.e. `collectionName`) if there is any data in that collection. This is the default behavior if you only specify a collection name in the `out` field.
- **merge** – Optional. Specify a collection name (e.g. `{ out: { merge: collectionName } }`) where the map-reduce operation writes output to an existing collection (i.e. `collectionName`,) and only overwrites existing documents when a new document has the same key as an "old" document in this collection.

- **reduce** – Optional. This operation behaves as the `merge` option above, except that when an existing document has the same key as a new document, `reduce` function from the map reduce job will run on both values and MongoDB writes the result of this function to the new collection. The specification takes the form of `{ out: { reduce: collectionName } }`, where `collectionName` is the name of the results collection.
- **inline** – Optional. Indicate the inline option (i.e. `{ out: { inline: 1 } }`) to perform the map reduce job in ram and return the results at the end of the function. This option is only possible when the entire result set will fit within the *maximum size of a BSON document* (page 679). When performing map-reduce jobs on secondary members of replica sets, this is the only available option.
- **db** – Optional. The name of the database that you want the map-reduce operation to write its output. By default this will be the same database as the input collection.
- **sharded** – Optional. If `true`, *and* the output mode writes to a collection, and the output database has sharding enabled, the map-reduce operation will shard the results collection according to the `_id` field.
- **nonAtomic** – New in version 2.1. Optional. Specify output operation as non-atomic such that the output behaves like a normal `multi update()`. If `true`, the post processing step will not execute inside of a database lock so that partial results will be visible during processing . `nonAtomic` is valid only for `merge` and `reduce` output operations where post-processing may be a long-running operation.

See Also:

"`mapReduce()`" and "*map-reduce*."

Also, the "MapReduce page, provides a greater overview of MongoDB's map-reduce functionality. Consider the "Simple application" support for basic aggregation operations and "*Aggregation Framework* (page 253)" for a more flexible approach to data aggregation in MongoDB.

mapreduce.shardedfinish (internal)

`mapreduce.`**`shardedfinish`**

Provides internal functionality to support *map-reduce* in *sharded* environments.

See Also:

"`mapReduce`"

medianKey (internal)

`medianKey`

`medianKey` is an internal command.

migrateClone (internal)

`_migrateClone`

`_migrateClone` is an internal command. Do not call directly.

moveChunk

moveChunk

moveChunk is an internal administrative command that moves *chunks* between *shards*. You must issue the moveChunk command against the *admin database* in the form:

```
db.runCommand( { moveChunk : <namespace> ,
                 find : <query> ,
                 to : <destination>,
                 <options> } )
```

Parameters

- **moveChunk** – The name of the *collection* where the *chunk* exists. Specify the collection's full namespace, including the database name.
- **find** – A document including the *shard key*.
- **to** – The identifier of the shard, that you want to migrate the chunk to.
- **_secondaryThrottle** – Optional. Set to false by default. Provides *write concern* (page 54) support for chunk migrations.

If you set _secondaryThrottle to true, during chunk migrations when a *shard* hosted by a *replica set*, the mongod will wait until the *secondary* members replicate the migration operations continuing to migrate chunk data. You may also configure _secondaryThrottle in the balancer configuration.

Use the sh.moveChunk() helper in the mongo shell to migrate chunks manually.

The *chunk migration* (page 138) section describes how chunks move between shards on MongoDB.

moveChunk will return the following if another cursor is using the chunk you are moving:

```
errmsg: "The collection's metadata lock is already taken."
```

These errors usually occur when there are too many open *cursors* accessing the chunk you are migrating. You can either wait until the cursors complete their operation or close the cursors manually.

Note: Only use the moveChunk in special circumstances such as preparing your *sharded cluster* for an initial ingestion of data, or a large bulk import operation. See *Create Chunks (Pre-Splitting)* (page 121) for more information.

---

movePrimary

movePrimary

In a *sharded cluster*, this command reassigns the database's *primary shard*, which holds all un-sharded collections in the database. movePrimary is an administrative command that is only available for mongos (page 676) instances. Only use movePrimary when removing a shard from a sharded cluster.

Important: Only use movePrimary when:

- the database does not contain any collections with data, *or*
- you have drained all sharded collections using the removeShard command.

See *Remove Shards from an Existing Sharded Cluster* (page 145) for a complete procedure.

---

movePrimary changes the primary shard for this database in the cluster metadata, and migrates all un-sharded collections to the specified shard. Use the command with the following form:

```
{ movePrimary : "test", to : "shard0001" }
```

When the command returns, the database's primary location will shift to the designated *shard*. To fully decommission a shard, use the removeShard command.

netstat (internal)

netstat

netstat is an internal command that is only available on mongos (page 676) instances.

ping

ping

The ping command is a no-op used to test whether a server is responding to commands. This command will return immediately even if the server is write-locked:

```
{ ping: 1 }
```

The value (e.g. 1 above,) does not impact the behavior of the command.

printShardingStatus

printShardingStatus

Returns data regarding the status of a *sharded cluster* and includes information regarding the distribution of *chunks*. printShardingStatus is only available when connected to a *sharded cluster* via a mongos (page 676). Typically, you will use the sh.status() mongo shell wrapper to access this data.

profile

profile

Use the profile command to enable, disable, or change the query profiling level. This allows administrators to capture data regarding performance. The database profiling system can impact performance and can allow the server to write the contents of queries to the log, which might information security implications for your deployment. Consider the following prototype syntax:

```
{ profile: <level> }
```

The following profiling levels are available:

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

You may optionally set a threshold in milliseconds for profiling using the slowms option, as follows:

```
{ profile: 1, slowms: 200 }
```

mongod writes the output of the database profiler to the system.profile collection.

mongod records a record of queries that take longer than the slowms (page 627) to the log even when the database profiler is not active.

See Also:

Additional documentation regarding database profiling *Database Profiling* (page 174).

See Also:

"`db.getProfilingStatus()`" and "`db.setProfilingLevel()`" provide wrappers around this functionality in the `mongo` shell.

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)
```

Note: This command obtains a write lock on the affected database and will block other operations until it has completed; however the write lock is only in place while the enabling and disabling the profiler, which is typically a short operation.

---

reIndex

reIndex

The `reIndex` command rebuilds all indexes for a specified collection. Use the following syntax:

```
{ reIndex: "collection" }
```

Normally, MongoDB compacts indexes during routine updates. For most users, the `reIndex` is unnecessary. However, it may be worth running if the collection size has changed significantly or if the indexes are consuming a disproportionate amount of disk space.

Note that the `reIndex` command will block the server against writes and may take a long time for large collections.

Call `reIndex` using the following form:

```
db.collection.reIndex();
```

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

recvChunkAbort (internal)

_recvChunkAbort

`_recvChunkAbort` is an internal command. Do not call directly.

recvChunkCommit (internal)

_recvChunkCommit

`_recvChunkCommit` is an internal command. Do not call directly.

recvChunkStart (internal)

_recvChunkStart

_recvChunkStart is an internal command. Do not call directly.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

recvChunkStatus (internal)

_recvChunkStatus

_recvChunkStatus is an internal command. Do not call directly.

removeShard

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 migrating 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 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 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 , stage: "completed", host: "shardName", ok : 1 }
```

renameCollection

renameCollection

The renameCollection command is an administrative command that changes the name of an existing collection. You specify collections to renameCollection in the form of a complete *namespace*, which includes the database name. To rename a collection, issue the renameCollection command against the *admin database* in the form:

```
{ renameCollection: <source-namespace>, to: <target-namespace>[, dropTarget: <boolean> ] }
```

The `dropTarget` argument is optional.

If you specify a collection to the `to` argument in a different database, the `renameCollection` will copy the collection to the new database and then drop the source collection.

Parameters

- **source-namespace** – Specifies the complete namespace of the collection to rename.
- **to** (*string*) – Specifies the new namespace of the collection.
- **dropTarget** (*boolean*) – Optional. If `true`, `mongod` will drop the `target` of `renameCollection` prior to renaming the collection.

Exception

- **10026** – Raised if the `source` namespace does not exist.
- **10027** – Raised if the `target` namespace exists and `dropTarget` is either `false` or unspecified.
- **15967** – Raised if the `target` namespace is an invalid collection name.

You can use `renameCollection` in production environments; however:

- `renameCollection` will block all database activity for the duration of the operation.
- `renameCollection` is incompatible with sharded collections.

Warning: `renameCollection` will fail if *target* is the name of an existing collection and you do not specify `dropTarget: true`.
If the `renameCollection` operation does not complete the `target` collection and indexes will not be usable and will require manual intervention to clean up.

The shell helper `db.collection.renameCollection()` provides a simpler interface to using this command within a database. The following is equivalent to the previous example:

```
db.source-namespace.renameCollection( "target" )
```

Warning: You cannot use `renameCollection` with sharded collections.

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

repairDatabase

`repairDatabase`

Warning: In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` or related options like `db.repairDatabase()` in the `mongo` shell or *`mongod --repair`* (page 585). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase`. In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

The `repairDatabase` command checks and repairs errors and inconsistencies with the data storage. The command is analogous to a `fsck` command for file systems.

If your mongod instance is not running with journaling the system experiences an unexpected system restart or crash, and you have *no* other intact replica set members with this data, you should run the repairDatabase command to ensure that there are no errors in the data storage.

As a side effect, the repairDatabase command will compact the database, as the compact command, and also reduces the total size of the data files on disk. The repairDatabase command will also recreate all indexes in the database.

Use the following syntax:

```
{ repairDatabase: 1 }
```

Be aware that this command can take a long time to run if your database is large. In addition, it requires a quantity of free disk space equal to the size of your database. If you lack sufficient free space on the same volume, you can mount a separate volume and use that for the repair. In this case, you must run the command line and use the *--repairpath* (page 585) switch to specify the folder in which to store the temporary repair files.

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

This command is accessible via a number of different avenues. You may:

- Use the shell to run the above command, as above.
- Use the db.repairDatabase() in the mongo shell.
- Run mongod directly from your system's shell. Make sure that mongod isn't already running, and that you issue this command as a user that has access to MongoDB's data files. Run as:

  ```
  $ mongod --repair
  ```

  To add a repair path:

  ```
  $ mongod --repair --repairpath /opt/vol2/data
  ```

Note: This command will fail if your database is not a master or primary. In most cases, you should recover a corrupt secondary using the data from an existing intact node. If you must repair a secondary or slave node, first restart the node as a standalone mongod by omitting the *--replSet* (page 586) or *--slave* (page 587) options, as necessary.

---

replSetElect (internal)

replSetElect

replSetElect is an internal command that support replica set functionality.

replSetFreeze

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 (page 519) 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 the it against the *admin database*.

replSetFresh (internal)

`replSetFresh`

`replSetFresh` is an internal command that supports replica set functionality.

replSetGetRBID (internal)

`replSetGetRBID`

`replSetGetRBID` is an internal command that supports replica set functionality.

replSetGetStatus

`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 }
```

However, you can also run this command from the shell like so:

```
rs.status()
```

See Also:

"*Replica Set Status Reference* (page 659)" and "*Replication Fundamentals* (page 33)"

replSetHeartbeat (internal)

`replSetHeartbeat`

`replSetHeartbeat` is an internal command that supports replica set functionality.

replSetInitiate

`replSetInitiate`

The `replSetInitiate` command initializes a new replica set. Use the following syntax:

```
{ replSetInitiate : <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 661)," "*Replica Set Administration* (page 38)," and "*Replica Set Reconfiguration* (page 665)."

replSetMaintenance

`replSetMaintenance`

The `replSetMaintenance` 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: 1`, 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.

replSetReconfig

`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 :dbcommand'replSetReconfig' operation from occurring at the same time.

---

replSetStepDown

`replSetStepDown`

Options

- **force** (*boolean*) – Forces the *primary* to step down even if there aren't any secondary members within 10 seconds of the primary's latest optime. This option is not available in versions of `mongod` before 2.0.

The `replSetStepDown` (page 519) command forces the *primary* of the replica set to relinquish its status as primary. This initiates an *election for primary* (page 63). You may specify a number of seconds for the node to avoid election to primary:

```
{ replSetStepDown: <seconds> }
```

If you do not specify a value for `<seconds>`, `replSetStepDown` will attempt to avoid reelection to primary for 60 seconds.

Warning: This will force all clients currently connected to the database to disconnect. This help to ensure that clients maintain an accurate view of the replica set.

New in version 2.0: If there is no *secondary*, within 10 seconds of the primary, `replSetStepDown` (page 519) will not succeed to prevent long running elections.

replSetSyncFrom

replSetSyncFrom

New in version 2.2.

Options

- **host** – Specifies the name and port number of the set member that you want *this* member to sync from. Use the `[hostname]:[port]` form.

`replSetSyncFrom` allows you to explicitly configure which host the current `mongod` will poll *oplog* entries from. This operation may be useful for testing different patterns and in situations where a set member is not syncing from the host you want. You may **not** use this command to force a member to sync from:

- itself.
- an arbiter.
- 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 sync from a member that is more than 10 seconds behind the current member, `mongod` will return and log a warning, but *will* sync from such members.

The command has the following prototype form:

```
{ replSetSyncFrom: "[hostname]:[port]" }
```

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` provides a temporary override of default behavior. When you restart the `mongod` instance, it will revert to the default syncing logic. Always exercise caution with `replSetSyncFrom` when overriding the default behavior.

---

replSetTest (internal)

replSetTest

`replSetTest` is internal diagnostic command used for regression tests that supports replica set functionality.

resetError

resetError

The `resetError` command resets the last error status.

See Also:

`db.resetError()`

resync

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 no apply to replica sets.

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

serverStatus

serverStatus

The `serverStatus` command returns a document that provides an overview of the database process's state. Most monitoring applications run this command at a regular interval to collection statistics about the instance:

```
{ serverStatus: 1 }
```

The value (i.e. `1` above), does not affect the operation of the command.

See Also:

`db.serverStatus()` and "*Server Status Reference* (page 637)"

setParameter

setParameter

`setParameter` is an administrative command for modifying options normally set on the command line. You must issue the `setParameter` command against the *admin database* in the form:

```
{ setParameter: 1, <option>: <value> }
```

Replace the `<option>` with one of the following options supported by this command:

Options

- **journalCommitInterval** (*integer*) – Specify an integer between `1` and `500` signifying the number of milliseconds (ms) between journal commits.

  Consider the following example which sets the `journalCommitInterval` to `200` ms:

  ```
  use admin
  db.runCommand( { setParameter: 1, journalCommitInterval: 200 } )
  ```

  See Also:

  `journalCommitInterval` (page 625).

- **logLevel** (*integer*) – Specify an integer between `0` and `5` signifying the verbosity of the logging, where `5` is the most verbose.

  Consider the following example which sets the `logLevel` to `2`:

  ```
  use admin
  db.runCommand( { setParameter: 1, logLevel: 2 } )
  ```

  See Also:

  `verbose` (page 621).

- **notablescan** (*boolean*) – Specify whether queries must use indexes. If `true`, queries that perform a table scan instead of using an index will fail.

  Consider the following example which sets the `notablescan` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, notablescan: true } )
  ```

  See Also:

  `notablescan` (page 626).

- **traceExceptions** (*boolean*) – New in version 2.1. Configures `mongod` log full stack traces on assertions or errors. If `true`, `mongod` will log full stack traces on assertions or errors.

  Consider the following example which sets the `traceExceptions` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, traceExceptions: true } )
  ```

  See Also:

  `traceExceptions` (page 628).

- **quiet** (*boolean*) – Sets quiet logging mode. If `true`, `mongod` will go into a quiet logging mode which will not log the following events/activities:

  – connection events;

  – the `drop` command, the `dropIndexes` command, the `diagLogging` command, the `validate` command, and the `clean` command; and

  – replication synchronization activities.

  Consider the following example which sets the `quiet` to true:

  ```
  use admin
  db.runCommand( { setParameter: 1, quiet: true } )
  ```

  See Also:

  `quiet` (page 622).

- **syncdelay** (*integer*) – Specify the interval in seconds between *fsyncs* (i.e., flushes of memory to disk). By default, `mongod` will flush memory to disk every 60 seconds. Do not change this value unless you see a background flush average greater than 60 seconds.

  Consider the following example which sets the `syncdelay` to `60` seconds:

  ```
  use admin
  db.runCommand( { setParameter: 1, syncdelay: 60 } )
  ```

  See Also:

  `syncdelay` (page 627).

setShardVersion

`setShardVersion`

`setShardVersion` is an internal command that supports sharding functionality.

shardCollection

shardCollection

The `shardCollection` command marks a collection for sharding and will allow data to begin distributing among shards. You must run `enableSharding` on a database before running the `shardCollection` command.

```
{ shardCollection: "<db>.<collection>", key: { "<shardkey>": 1 } }
```

This enables sharding for the collection specified by `<collection>` in the database named `<db>`, using the key `<shardkey>` to distribute documents among the shard.

Choosing the right shard key to effectively distribute load among your shards requires some planning.

See Also:

Sharding (page 105) for more information related to sharding. Also consider the section on *Shard Keys* (page 109) for documentation regarding shard keys.

Warning: There's no easy way to disable sharding after running `shardCollection`. In addition, you cannot change shard keys once set. If you must convert a sharded cluster to a *standalone* node or *replica set*, you must make a single backup of the entire cluster and then restore the backup to the standalone `mongod` or the replica set..

shardingState

shardingState

The `shardingState` command returns `true` if the `mongod` instance is a member of a sharded cluster. Run the command using the following syntax:

```
{ shardingState: 1 }
```

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.

shutdown

shutdown

The `shutdown` command cleans up all database resources and then terminates the process. You must issue the `shutdown` command against the *admin database* in the form:

```
{ shutdown: 1 }
```

Note: Run the `shutdown` against the *admin database*. When using `shutdown`, the connection must originate from localhost **or** use an authenticated connection.

If the node you're trying to shut down is a *replica set* (page 33) primary, then the command will succeed only if there exists a secondary node whose oplog data is within 10 seconds of the primary. You can override this protection using the `force` option:

```
{ shutdown: 1, force: true }
```

Alternatively, the `shutdown` command also supports a `timeoutSecs` argument which allows you to specify a number of seconds to wait for other members of the replica set to catch up:

```
{ shutdown: 1, timeoutSecs: 60 }
```

The equivalent `mongo` shell helper syntax looks like this:

```
db.shutdownServer({timeoutSecs: 60});
```

skewClockCommand (internal)

_skewClockCommand

`_skewClockCommand` is an internal command. Do not call directly.

sleep (internal)

sleep

`sleep` is an internal command for testing purposes. The `sleep` command forces the database to block all operations. It takes the following options:

Parameters

- **w** (*boolean*) – If true, obtain a global write lock. Otherwise obtains a read lock.
- **secs** (*integer*) – Specifies the number of seconds to sleep.

```
{ sleep: { w: true, secs: <seconds> } }
```

The above command places the `mongod` instance in a "write-lock" state for a specified (i.e. `<seconds>`) number of seconds. Without arguments, `sleep`, causes a "read lock" for 100 seconds.

Warning: `sleep` claims the lock specified in the `w` argument and blocks *all* operations on the `mongod` instance for the specified amount of time.

split

split

The `split` command creates new *chunks* in a *sharded* environment. While splitting is typically managed automatically by the `mongos` (page 676) instances, this command makes it possible for administrators to manually create splits.

In normal operation there is no need to manually split chunks

The *balancer* and other sharding infrastructure will automatically create chunks in the course of normal operations. See *Sharding Internals* (page 133) for more information.

Consider the following example:

```
db.runCommand( { split : "test.people" , find : { _id : 99 } } )
```

This command inserts a new split in the collection named `people` in the `test` database. This will split the chunk that contains the document that matches the query `{ _id : 99 }` in half. If the document specified by the query does not (yet) exist, the `split` will divide the chunk where that document *would* exist.

The split divides the chunk in half, and does *not* split the chunk using the identified document as the middle. To define an arbitrary split point, use the following form:

```
db.runCommand( { split : "test.people" , middle : { _id : 99 } } )
```

This form is typically used when *pre-splitting* data in a collection.

`split` is an administrative command that is only available for `mongos` (page 676) instances.

splitChunk

splitChunk

`splitChunk` is an internal command. Use the `sh.splitFind()` and `sh.splitAt()` functions in the `mongo` shell to access this functionality.

testDistLockWithSkew (internal)

_testDistLockWithSkew

`_testDistLockWithSkew` is an internal command. Do not call directly.

testDistLockWithSyncCluster (internal)

_testDistLockWithSyncCluster

`_testDistLockWithSyncCluster` is an internal command. Do not call directly.

top

top

The `top` command is an administrative command which returns raw usage of each database, and provides amount of time, in microseconds, used and a count of operations for the following event types:

- total
- readLock
- writeLock
- queries
- getmore
- insert
- update
- remove
- commands

You must issue the `top` command against the *admin database* in the form:

```
{ top: 1 }
```

touch

touch

New in version 2.2. The `touch` command loads data from the data storage layer into memory. `touch` can load the data (i.e. documents,) indexes or both documents and indexes. Use this command to ensure that a collection, and/or its indexes, are in memory before another operation. By loading the collection or indexes into memory, `mongod` will ideally be able to perform subsequent operations more efficiently. The `touch` command has the following prototypical form:

```
{ touch: [collection], data: [boolean], index: [boolean] }
```

By default, `data` and `index` are false, and `touch` will perform no operation. For example, to load both the data and the index for a collection named `records`, you would use the following command in the `mongo` shell:

```
db.runCommand({ touch: "records", data: true, index: true })
```

`touch` will not block read and write operations on a `mongod`, and can run on *secondary* members of replica sets.

Note: Using `touch` to control or tweak what a `mongod` stores in memory may displace other records data in memory and hinder performance. Use with caution in production systems.

---

transferMods (internal)

_transferMods

`_transferMods` is an internal command. Do not call directly.

unsetSharding (internal)

unsetSharding

`unsetSharding` is an internal command that supports sharding functionality.

validate

validate

The `validate` command checks the contents of a namespace by scanning a collection's data and indexes for correctness. The command can be slow, particularly on larger data sets:

```
{ validate: "users" }
```

This command will validate the contents of the collection named `users`. You may also specify one of the following options:

- `full:` `true` provides a more thorough scan of the data.
- **`scandata:` `false` skips the scan of the base collection** without skipping the scan of the index.

The `mongo` shell also provides a wrapper:

```
db.collection.validate();
```

Use one of the following forms to perform the full collection validation:

```
db.collection.validate(true)
db.runCommand( { validate: "collection", full: true } )
```

Warning: This command is resource intensive and may have an impact on the performance of your MongoDB instance.

whatsmyuri (internal)

whatsmyuri

`whatsmyuri` is an internal command.

writebacklisten (internal)

writebacklisten

`writebacklisten` is an internal command.

writeBacksQueued (internal)

writeBacksQueued

`writeBacksQueued` is an internal command that returns true if there are operations in the write back queue for the given `mongos` (page 676). This command applies to *sharded clusters* only.

36.1.3 JavaScript Methods

Date()

Date

Returns Current date, as a string.

ObjectId.getTimestamp()

`ObjectId.`**getTimestamp**

Returns The timestamp portion of the *ObjectId()* (page 288) object as a Date.

In the following example, call the `getTimestamp()` (page 527) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`), as follows:

```
ObjectId("507c7f79bcf86cd7994f6c0e").getTimestamp()
```

This will return the following output:

```
ISODate("2012-10-15T21:26:17Z")
```

ObjectId.toString()

`ObjectId.`**toString**

Returns The string representation of the *ObjectId()* (page 288) object. This value has the format of `ObjectId(...)`.

Changed in version 2.2: In previous versions `ObjectId.toString()` (page 527) returns the value of the ObjectId as a hexadecimal string. In the following example, call the `toString()` (page 527) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`), as follows:

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

This will return the following string:

```
ObjectId("507c7f79bcf86cd7994f6c0e")
```

You can confirm the type of this object using the following operation:

```
typeof ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

ObjectId.valueOf()

`ObjectId.valueOf`

Returns The value of the *ObjectId()* (page 288) object as a lowercase hexadecimal string. This value is the `str` attribute of the `ObjectId()` object.

Changed in version 2.2: In previous versions `ObjectId.valueOf()` (page 528) returns the the `ObjectId()` object. In the following example, call the `valueOf()` (page 528) method on an ObjectId (e.g. `ObjectId("507c7f79bcf86cd7994f6c0e")`), as follows:

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

This will return the following string:

```
507c7f79bcf86cd7994f6c0e
```

You can confirm the type of this object using the following operation:

```
typeof ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

cat()

cat

Parameters

- **filename** (*string*) – Specify a path and file name on the local file system.

Returns the contents of the specified file.

This function returns with output relative to the current shell session, and does not impact the server.

cd()

cd

Parameters

- **file** (*string*) – Specify a path on the local file system.

Changes the current working directory to the specified path.

This function returns with output relative to the current shell session, and does not impact the server.

Note: This feature is not yet implemented.

clearRawMongoProgramOutput()

clearRawMongoProgramOutput
For internal use.

copyDbpath()

copyDbpath
For internal use.

cursor.batchSize()

cursor.**batchSize**
The `batchSize()` (page 529) method specifies the number of documents to return in each batch of the response from the MongoDB instance. In most cases, modifying the batch size will not affect the user or the application since the `mongo` shell and most *drivers* (page 285) return results as if MongoDB returned a single batch.

The `batchSize()` (page 529) method takes the following parameter:

Parameters

- **size** – The number of documents to return per batch. Do **not** use a batch size of `1`.

Note: Specifying `1` or a negative number is analogous to using the `limit()` method.

Consider the following example of the `batchSize()` (page 529) method in the `mongo` shell:

```
db.inventory.find().batchSize(10)
```

This operation will set the batch size for the results of a query (i.e. `find()`) to `10`. The effects of this operation do not affect the output in the `mongo` shell, which always iterates over the first 20 documents.

cursor.count()

cursor.**count**

Parameters

- **override** (*boolean*) – Override the effects of the `cursor.skip()` and `cursor.limit()` methods on the cursor.

Append the `count()` method on a "`find()`" query to return the number of matching objects for any query.

In normal operation, `cursor.count()` ignores the effects of the `cursor.skip()` and `cursor.limit()`. To consider these effects specify "`count(true)`".

See Also:

`cursor.size()`.

cursor.explain()

`cursor.explain`

Returns A document that describes the process used to return the query.

This method may provide useful insight when attempting to optimize a query. When you call the `explain` on a query, the query system reevaluates the query plan. As a result, these operations only provide a realistic account of *how* MongoDB would perform the query, and *not* how long the query would take. The method's output includes these fields:

- `cursor`: The value for cursor can be either `BasicCursor` or `BtreeCursor`. The second of these indicates that the given query is using an index.
- `nscanned`: The number of index entries scanned.
- `n`: the number of documents returned by the query. You want the value of `n` to be close to the value of `nscanned`. You want to avoid a "collection scan," which is a scan where every document in the collection is accessed. This is the case when `nscanned` is equal to the number of documents in the collection.
- `millis`: the number of milliseconds require to complete the query. This value is useful for comparing indexing strategies.

See Also:

`$explain` (page 448) for related functionality and the "Optimization" wiki page for information regarding optimization strategies.

cursor.forEach()

`cursor.forEach`

Parameters

- **function** – function to apply to each document visited by the cursor.

Provides the ability to loop or iterate over the cursor returned by a `db.collection.find()` query and returns each result on the shell. Specify a JavaScript function as the argument for the `cursor.forEach()` function. Consider the following example:

```
db.users.find().forEach( function(u) { print("user: " + u.name); } );
```

See Also:

`cursor.map()` for similar functionality.

cursor.hasNext()

`cursor.hasNext`

Returns Boolean.

`cursor.hasNext()` returns `true` if the cursor returned by the `db.collection.find()` query can iterate further to return more documents.

cursor.hint()

`cursor.hint`

Arguments

- **index** – The specification for the index to "hint" or force MongoDB to use when performing the query.

Call this method on a query to override MongoDB's default index selection and query optimization process. The argument is an index specification, like the argument to `ensureIndex()`. Use `db.collection.getIndexes()` to return the list of current indexes on a collection.

See Also:

"`$hint` (page 449)"

cursor.limit()

`cursor.`**`limit`**

Use the `cursor.limit()` method on a cursor to specify the maximum number of documents a the cursor will return. `cursor.limit()` is analogous to the `LIMIT` statement in a SQL database.

Note: You must apply `cursor.limit()` to the cursor before retrieving any documents from the database.

Use `cursor.limit()` to maximize performance and prevent MongoDB from returning more results than required for processing.

A `cursor.limit()` value of 0 (e.g. "`.limit(0)`") is equivalent to setting no limit.

cursor.map()

`cursor.`**`map`**

Parameters

- **function** – function to apply to each document visited by the cursor.

Apply *function* to each document visited by the cursor, and collect the return values from successive application into an array. Consider the following example:

```
db.users.find().map( function(u) { return u.name; } );
```

See Also:

`cursor.forEach()` for similar functionality.

cursor.next()

`cursor.`**`next`**

Returns The next document in the cursor returned by the `db.collection.find()` method. See `cursor.hasNext()` related functionality.

cursor.readPref()

`cursor.`**`readPref`**

Parameters

- **mode** (*string*) – Read preference mode
- **tagSet** (*array*) – Optional. Array of tag set objects

Append the `readPref()` to a cursor to control how the client will route the query will route to members of the replica set.

The `mode` string should be one of:

- `primary` (page 56)
- `primaryPreferred` (page 57)
- `secondary` (page 57)
- `secondaryPreferred` (page 57)
- `nearest` (page 57)

The `tagSet` parameter, if given, should consist of an array of tag set objects for filtering secondary read operations. For example, a secondary member tagged `{ dc: 'ny', rack: 2, size: 'large' }` will match the tag set `{ dc: 'ny', rack: 2 }`. Clients match tag sets first in the order they appear in the read preference specification. You may specify an empty tag set `{}` as the last element to default to any available secondary. See the:ref:*tag sets <replica-set-read-preference-tag-sets>* documentation for more information.

Note: You must apply `cursor.readPref()` to the cursor before retrieving any documents from the database.

---

cursor.showDiskLoc()

`cursor.`**`showDiskLoc`**

Returns A modified cursor object that contains documents with appended information that describes the on-disk location of the document.

See Also:

`$showDiskLoc` (page 464) for related functionality.

cursor.size()

`cursor.`**`size`**

Returns A count of the number of documents that match the `db.collection.find()` query after applying any `cursor.skip()` and `cursor.limit()` methods.

cursor.skip()

`cursor.`**`skip`**

Call the `cursor.skip()` method on a cursor to control where MongoDB begins returning results. This approach may be useful in implementing "paged" results.

Note: You must apply `cursor.skip()` to the cursor before retrieving any documents from the database.

Consider the following JavaScript function as an example of the sort function:

```
function printStudents(pageNumber, nPerPage) {
   print("Page: " + pageNumber);
   db.students.find().skip((pageNumber-1)*nPerPage).limit(nPerPage).forEach( function(student) {
}
```

The `cursor.skip()` method is often expensive because it requires the server to walk from the beginning of the collection or index to get the offset or skip position before beginning to return result. As offset (e.g. `pageNumber` above) increases, `cursor.skip()` will become slower and more CPU intensive. With larger collections, `cursor.skip()` may become IO bound.

Consider using range-based pagination for these kinds of tasks. That is, query for a range of objects, using logic within the application to determine the pagination rather than the database itself. This approach features better index utilization, if you do not need to easily jump to a specific page.

cursor.snapshot()

`cursor.`**`snapshot`**

Append the `cursor.snapshot()` method to a cursor to toggle the "snapshot" mode. This ensures that the query will not miss any documents and return no duplicates, even if other operations modify objects while the query runs.

Note: You must apply `cursor.snapshot()` to the cursor before retrieving any documents from the database.

Queries with results of less than 1 megabyte are effectively implicitly snapshotted.

cursor.sort()

`cursor.`**`sort`**

Parameters

- **sort** – A document whose fields specify the attributes on which to sort the result set.

Append the `sort()` method to a cursor to control the order that the query returns matching documents. For each field in the sort document, if the field's corresponding value is positive, then `sort()` returns query results in ascending order for that attribute: if the field's corresponding value is negative, then `sort()` returns query results in descending order.

Note: You must apply `cursor.limit()` to the cursor before retrieving any documents from the database.

Consider the following example:

```
db.collection.find().sort( { age: -1 } );
```

Here, the query returns all documents in `collection` sorted by the `age` field in descending order. Specify a value of negative one (e.g. `-1`), as above, to sort in descending order or a positive value (e.g. `1`) to sort in ascending order.

Unless you have a index for the specified key pattern, use `cursor.sort()` in conjunction with `cursor.limit()` to avoid requiring MongoDB to perform a large, in-memory sort. `cursor.limit()` increases the speed and reduces the amount of memory required to return this query by way of an optimized algorithm.

Warning: The sort function requires that the entire sort be able to complete within 32 megabytes. When the sort option consumes more than 32 megabytes, MongoDB will return an error. Use `cursor.limit()`, or create an index on the field that you're sorting to avoid this error.

The `$natural` parameter returns items according to their order on disk. Consider the following query:

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

This will return documents in the reverse of the order on disk. Typically, the order of documents on disks reflects insertion order, *except* when documents move internal because of document growth due to update operations.

db.addUser()

`db.addUser`

Parameters

- **username** (*string*) – Specifies a new username.
- **password** (*string*) – Specifies the corresponding password.
- **readOnly** (*boolean*) – Optional. Restrict a user to read-privileges only. Defaults to false.

Use this function to create new database users, by specifying a username and password as arguments to the command. If you want to restrict the user to have only read-only privileges, supply a true third argument; however, this defaults to false.

db.auth()

`db.auth`

Parameters

- **username** (*string*) – Specifies an existing username with access privileges for this database.
- **password** (*string*) – Specifies the corresponding password.

Allows a user to authenticate to the database from within the shell. Alternatively use *mongo --username* (page 592) and *--password* (page 592) to specify authentication credentials.

db.cloneDatabase()

`db.cloneDatabase`

Parameters

- **hostname** (*string*) – Specifies the hostname to copy the current instance.

Use this function to copy a database from a remote to the current database. The command assumes that the remote database has the same name as the current database. For example, to clone a database named `importdb` on a host named `hostname`, do

```
use importdb
db.cloneDatabase("hostname");
```

New databases are implicitly created, so the current host does not need to have a database named `importdb` for this command to succeed.

This function provides a wrapper around the MongoDB *database command* "`clone`." The `copydb` database command provides related functionality.

db.collection.aggregate()

`db.collection.`**`aggregate`**

New in version 2.1.0. Always call the `db.collection.aggregate()` method on a collection object.

Arguments

- **pipeline** – Specifies a sequence of data aggregation processes. See the *aggregation reference* (page 269) for documentation of these operators.

Consider the following example from the *aggregation documentation* (page 253).

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : { tags : 1 },
     authors : { $addToSet : "$author" }
  } }
);
```

See Also:

"`aggregate`," "*Aggregation Framework* (page 253)," and "*Aggregation Framework Reference* (page 269)."

db.collection.dataSize()

`db.collection.`**`dataSize`**

Returns The size of the collection. This method provides a wrapper around the `size` (page 654) output of the `collStats` (i.e. `db.collection.stats()`) command.

db.collection.distinct()

`db.collection.`**`distinct`**

Parameters

- **string** (*field*) – A field that exists in a document or documents within the *collection*.

Returns an array that contains a list of the distinct values for the specified field.

Note: The `db.collection.distinct()` method provides a wrapper around the `distinct`. Results must not be larger than the maximum *BSON size* (page 679).

---

db.collection.drop()

`db.collection.`**`drop`**

Call the `db.collection.drop()` method on a collection to drop it from the database.

db.collection.drop() takes no arguments and will produce an error if called with any arguments.

db.collection.dropIndex()

db.collection.**dropIndex**

Drops or removes the specified index from a collection. The db.collection.dropIndex() method provides a wrapper around the dropIndexes command.

The db.collection.dropIndex() method takes the following parameter:

Parameters

- **index** – Specifies either the name or the key of the index to drop. You **must** use the name of the index if you specified a name during the index creation.

The db.collection.dropIndex() method cannot drop the _id index. Use the db.collection.getIndexes() method to view all indexes on a collection.

Consider the following examples of the db.collection.dropIndex() method that assumes the following indexes on the collection pets:

```
> db.pets.getIndexes()
[
   {  "v" : 1,
      "key" : { "_id" : 1 },
      "ns" : "test.pets",
      "name" : "_id_"
   },
   {
      "v" : 1,
      "key" : { "cat" : -1 },
      "ns" : "test.pets",
      "name" : "catIdx"
   },
   {
      "v" : 1,
      "key" : { "cat" : 1, "dog" : -1 },
      "ns" : "test.pets",
      "name" : "cat_1_dog_-1"
   }
]
```

•To drop the index on the field cat, you must use the index name catIdx:

```
db.pets.dropIndex( 'catIdx' )
```

•To drop the index on the fields cat and dog, you use either the index name cat_1_dog_-1 or the key { "cat" : 1, "dog" : -1 }:

```
db.pets.dropIndex( 'cat_1_dog_-1' )

db.pets.dropIndex( { cat : 1, dog : -1 } )
```

db.collection.dropIndexes()

db.collection.**dropIndexes**

Drops all indexes other than the required index on the _id field. Only call dropIndexes() as a method on a collection object.

db.collection.ensureIndex()

`db.collection.ensureIndex`

Parameters

- **keys** (*document*) – A *document* that contains pairs with the name of the field or fields to index and order of the index. A `1` specifies ascending and a `-1` specifies descending.
- **options** (*document*) – A *document* that controls the creation of the index. This argument is optional.

> **Warning:** Index names, including their full namespace (i.e. `database.collection`) can be no longer than 128 characters. See the `db.collection.getIndexes()` field "`name`" for the names of existing indexes.

Creates an index on the field specified, if that index does not already exist. If the `keys` document specifies more than one field, than `db.collection.ensureIndex()` creates a *compound index*. For example:

```
db.collection.ensureIndex({ [key]: 1})
```

This command creates an index, in ascending order, on the field `[key]`. To specify a compound index use the following form:

```
db.collection.ensureIndex({ [key]: 1, [key1]: -1 })
```

This command creates a compound index on the `key` field (in ascending order) and `key1` field (in descending order.)

Note: Typically the order of an index is only important when doing `cursor.sort()` operations on the indexed fields.

The available options, possible values, and the default settings are as follows:

Option	Value	Default
background	true or false	false
unique	true or false	false
name	string	none
cache	true or false	true
dropDups	true or false	false
sparse	true or false	false
expireAfterSeconds	integer	none
v	index version.	1 [1]

Options

[1] The default index version depends on the version of `mongod` running when creating the index. Before version 2.0, the this value was 0; versions 2.0 and later use version 1.

- **background** (*Boolean*) – Specify `true` to build the index in the background so that building an index will *not* block other database activities.
- **unique** (*Boolean*) – Specify `true` to create a unique index so that the collection will not accept insertion of documents where the index key or keys matches an existing value in the index.
- **name** (*string*) – Specify the name of the index. If unspecified, MongoDB will generate an index name by concatenating the names of the indexed fields and the sort order.
- **cache** (*Boolean*) – Specify `false` to prevent caching of this `db.collection.ensureIndex()` call in the index cache.
- **dropDups** (*Boolean*) – Specify `true` when creating a unique index, on a field that *may* have duplicate to index only the first occurrence of a key, and ignore subsequent occurrences of that key.
- **sparse** (*Boolean*) – If `true`, the index only references documents with the specified field. These indexes use less space, but behave differently in some situations (particularly sorts.)
- **expireAfterSeconds** (*integer*) – Specify a value, in seconds, as a *TTL* to control how long MongoDB will retain documents in this collection. See "*Expire Data from Collections by Setting TTL* (page 296)" for more information on this functionality.
- **v** – Only specify a different index version in unusual situations. The latest index version (version 1) provides a smaller and faster index format.

Please be aware of the following behaviors of `ensureIndex()`:

- To add or change index options you must drop the index using the `db.collection.dropIndex()` and issue another `ensureIndex()` operation with the new options.

  If you create an index with one set of options, and then issue `ensureIndex()` method command with the same index fields and different options without first dropping the index, `ensureIndex()` will *not* rebuild the existing index with the new options.

- If you call multiple `ensureIndex()` methods with the same index specification at the same time, only the first operation will succeed, all other operations will have no effect.
- Non-background indexing operations will block all other operations on a database.
- You cannot stop a foreground index build once it's begun. See the *Monitoring and Controlling Index Building* (page 242) for more information.

db.collection.find()

`db.collection.find`

The `find()` method selects documents in a collection and returns a *cursor* to the selected documents.

The `find()` method takes the following parameters.

Parameters

- **query** (*document*) – Optional. Specifies the selection criteria using `query operators`. Omit the `query` parameter or pass an empty document (e.g. `{}`) to return all documents in the collection.
- **projection** (*document*) – Optional. Controls the fields to return, or the *projection*. The `projection` argument will resemble the following prototype:

```
{ field1: boolean, field2: boolean ... }
```

The `boolean` can take the following include or exclude values:

- `1` or `true` to include. The `find()` method always includes the *_id* field even if the field is not explicitly stated to return in the *projection* parameter.
- `0` or `false` to exclude.

The `projection` cannot contain both include and exclude specifications except for the exclusion of the `_id` field.

Returns A *cursor* to the documents that match the `query` criteria and contain the `projection` fields.

Note: In the `mongo` shell, you can access the returned documents directly without explicitly using the JavaScript cursor handling method. Executing the query directly on the `mongo` shell prompt automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

Consider the following examples of the `find()` method:

- To select all documents in a collection, call the `find()` method with no parameters:

```
db.products.find()
```

This operation returns all the documents with all the fields from the collection `products`. By default, in the `mongo` shell, the cursor returns the first batch of 20 matching documents. In the `mongo` shell, iterate through the next batch by typing `it`. Use the appropriate cursor handling mechanism for your specific language driver.

- To select the documents that match a selection criteria, call the `find()` method with the `query` criteria:

```
db.products.find( { qty: { $gt: 25 } } )
```

This operation returns all the documents from the collection `products` where `qty` is greater than `25`, including all fields.

- To select the documents that match a selection criteria and return, or *project* only certain fields into the result set, call the `find()` method with the `query` criteria and the `projection` parameter, as in the following example:

```
db.products.find( { qty: { $gt: 25 } }, { item: 1, qty: 1 } )
```

This operation returns all the documents from the collection `products` where `qty` is greater than `25`. The documents in the result set only include the `_id`, `item`, and `qty` fields using "inclusion" projection. `find()` always returns the `_id` field, even when not explicitly included:

```
{ "_id" : 11, "item" : "pencil", "qty" : 50 }
{ "_id" : ObjectId("50634d86be4617f17bb159cd"), "item" : "bottle", "qty" : 30 }
{ "_id" : ObjectId("50634dbcbe4617f17bb159d0"), "item" : "paper", "qty" : 100 }
```

- To select the documents that match a query criteria and exclude a set of fields from the resulting documents, call the `find()` method with the `query` criteria and the `projection` parameter using the `exclude` syntax:

```
db.products.find( { qty: { $gt: 25 } }, { _id: 0, qty: 0 } )
```

The query will return all the documents from the collection `products` where `qty` is greater than `25`. The documents in the result set will contain all fields *except* the `_id` and `qty` fields, as in the following:

```
{ "item" : "pencil", "type" : "no.2" }
{ "item" : "bottle", "type" : "blue" }
{ "item" : "paper" }
```

db.collection.findAndModify()

db.collection.**findAndModify**

The `db.collection.findAndModify()` method atomically modifies and returns a single document. By default, the returned document does not include the modifications made on the update. To return the document with the modifications made on the update, use the `new` option.

The `db.collection.findAndModify()` method takes a document parameter with the following sub-document fields:

Fields

- **query** (*document*) – Optional. Specifies the selection criteria for the modification. The `query` field employs the same *query selectors* as used in the `db.collection.find()` method. Although the query may match multiple documents, `findAndModify()` will only select one document to modify.

  The `query` field has the following syntax:

  ```
  query: { <query expression> }
  ```

- **sort** (*document*) – Optional. Determines which document the operation will modify if the query selects multiple documents. `findAndModify()` will modify the first document in the sort order specified by this argument.

  The `sort` field has the following syntax:

  ```
  sort: { field1: value1, field2: value2, ... }
  ```

- **remove** (*boolean*) – Optional if `update` field exists. When `true`, removes the selected document. The default is `false`.

  The `remove` field has the following syntax:

  ```
  remove: <boolean>
  ```

- **update** (*document*) – Optional if `remove` field exists. Performs an update of the selected document. The `update` field employs the same *update operators* or `field: value` specifications to modify the selected document.

  ```
  update: { <update expression> }
  ```

- **new** (*boolean*) – Optional. When `true`, returns the modified document rather than the original. The `findAndModify` method ignores the `new` option for `remove` operations. The default is `false`.

  ```
  new: <boolean>
  ```

- **fields** (*document*) – Optional. A subset of fields to return.

  ```
  fields: { field1: <boolean>, field2: <boolean> ... }
  ```

- **upsert** (*boolean*) – Optional. Used in conjunction with the `update` field. When `true`, `findAndModify` creates a new document if the `query` returns no documents. The default is `false`. In version 2.2, the `findAndModify` command returns `null` when `upsert` is `true`.

  ```
  upsert: <boolean>
  ```

Consider the following example:

```
db.people.findAndModify( {
    query: { name: "Tom", state: "active", rating: { $gt: 10 } },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } }
    } );
```

This command performs the following actions:

1. The `query` finds a document in the `people` collection where the `name` field has the value `Tom`, the `state` field has the value `active` and the `rating` field has a value : operator:*greater than* <*$gt*> 10.
2. The `sort` orders the results of the query in ascending order.
3. The update `increments` the value of the `score` field by 1.
4. The command returns the original unmodified document selected for this update.

Warning: When using `findAndModify` in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` operations issued against `mongos` (page 676) instances for non-sharded collections function normally.

db.collection.findOne()

`db.collection.`**`findOne`**

Parameters

- **query** (*document*) – Optional. A *document* that specifies the *query* using the JSON-like syntax and `query operators`.

Returns One document that satisfies the query specified as the argument to this method.

Returns only one document that satisfies the specified query. If multiple documents satisfy the query, this method returns the first document according to the *natural order* which reflects the order of documents on the disc. In *capped collections*, natural order is the same as insertion order.

db.collection.getIndexes()

`db.collection.`**`getIndexes`**

Returns an array that holds a list of documents that identify and describe the existing indexes on the collection. You must call the `db.collection.getIndexes()` on a collection. For example:

```
db.collection.getIndexes()
```

Change `collection` to the name of the collection whose indexes you want to learn.

The `db.collection.getIndexes()` items consist of the following fields:

`getIndexes.v`
Holds the version of the index.

The index version depends on the version of `mongod` that created the index. Before version 2.0 of MongoDB, the this value was 0; versions 2.0 and later use version 1.

`getIndexes.key`
Contains a document holding the keys held in the index, and the order of the index. Indexes may be either descending or ascending order. A value of negative one (e.g. `-1`) indicates an index sorted in descending order while a positive value (e.g. `1`) indicates an index sorted in an ascending order.

`getIndexes.ns`
The namespace context for the index.

`getIndexes.name`
A unique name for the index comprised of the field names and orders of all keys.

db.collection.group()

`db.collection.group`
The `db.collection.group()` accepts a single *document* that contains the following:

Fields

- **key** – Specifies one or more fields to group by.
- **reduce** – Specifies a reduce function that operates over all the iterated objects. Typically these aggregator functions perform some sort of summing or counting. The reduce function takes two arguments: the current document and an aggregation counter object.
- **initial** – The starting value of the aggregation counter object.
- **keyf** – Optional. An optional function that returns a "key object" for use as the grouping key. Use `keyf` instead of `key` to specify a key that is not a single/multiple existing fields. For example, use `keyf` to group by day or week in place of a fixed `key`.
- **cond** – Optional. A statement that must evaluate to true for the `db.collection.group()` to process this document. Simply, this argument specifies a query document (as for `db.collection.find()`). Unless specified, `db.collection.group()` runs the "reduce" function against all documents in the collection.
- **finalize** – Optional. A function that runs each item in the result set before `db.collection.group()` returns the final value. This function can either modify the document by computing and adding an average field, or return compute and return a new document.

Warning: `db.collection.group()` does not work in *shard environments*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.

Note: The result set of the `db.collection.group()` must fit within a single *BSON* object.

Furthermore, you must ensure that there are fewer then 10,000 unique keys. If you have more than this, use `mapReduce`.

`db.collection.group()` provides a simple aggregation capability similar to the function of `GROUP BY` in SQL statements. Use `db.collection.group()` to return counts and averages from collections of MongoDB documents. Consider the following example `db.collection.group()` command:

```
db.collection.group(
              {key: { a:true, b:true },
               cond: { active: 1 },
               reduce: function(obj,prev) { prev.csum += obj.c; },
               initial: { csum: 0 }
              });
```

This command in the `mongo` shell groups the documents in the collection where the field `active` equals `1` into sets for all combinations of combinations values of the `a` and `b` fields. Then, within each group, the reduce function adds up each document's c field into the csum field in the aggregation counter document. This is equivalent to the following SQL statement.

```
SELECT a,b,sum(c) csum FROM collection WHERE active=1 GROUP BY a,b
```

See Also:

The "Aggregation" wiki page and "*Aggregation Framework* (page 253)."

db.collection.insert()

db.collection.**insert**

The `insert()` method inserts a document or documents into a collection. Changed in version 2.2: The `insert()` method can accept an array of documents to perform a bulk insert of the documents into the collection. Consider the following behaviors of the `insert()` method:

- If the collection does not exist, then the `insert()` method will create the collection.
- If the document does not specify an *_id* field, then MongoDB will add the `_id` field and assign a unique *ObjectId* for the document before inserting. Most drivers create an ObjectId and insert the `_id` field, but the `mongod` will create and populate the `_id` if the driver or application does not.
- If the document specifies a new field, then the `insert()` method inserts the document with the new field. This requires no changes to the data model for the collection or the existing documents.

The `insert()` method takes one of the following parameters:

Parameters

- **document** – A document to insert into the collection.
- **documents** (*array*) – New in version 2.2. An array of documents to insert into the collection.

Consider the following examples of the `insert()` method:

- To insert a single document and have MongoDB generate the unique `_id`, omit the `_id` field in the document and pass the document to the `insert()` method as in the following:

```
db.products.insert( { item: "card", qty: 15 } )
```

This operation inserts a new document into the `products` collection with the `item` field set to `card`, the `qty` field set to `15`, and the `_id` field set to a unique `ObjectId`:

```
{ "_id" : ObjectId("5063114bd386d8fadbd6b004"), "item" : "card", "qty" : 15 }
```

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`.

- To insert a single document, with a custom _id field, include the _id field set to a unique identifier and pass the document to the insert() method as follows:

```
db.products.insert( { _id: 10, item: "box", qty: 20 } )
```

This operation inserts a new document in the products collection with the _id field set to 10, the item field set to box, the qty field set to 20:

```
{ "_id" : 10, "item" : "box", "qty" : 20 }
```

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.

- To insert multiple documents, pass an array of documents to the insert() method as in the following:

```
db.products.insert( [ { _id: 11, item: "pencil", qty: 50, type: "no.2" },
                      {          item: "pen", qty: 20 },
                      {          item: "eraser", qty: 25 } ] )
```

The operation will insert three documents into the products collection:

  - A document with the fields _id set to 11, item set to pencil, qty set to 50, and the type set to no.2.
  - A document with the fields _id set to a unique objectid, item set to pen, and qty set to 20.
  - A document with the fields _id set to a unique objectid, item set to eraser, and qty set to 25.

```
{ "_id" : 11, "item" : "pencil", "qty" : 50, "type" : "no.2" }
{ "_id" : ObjectId("50631bc0be4617f17bb159ca"), "item" : "pen", "qty" : 20 }
{ "_id" : ObjectId("50631bc0be4617f17bb159cb"), "item" : "eraser", "qty" : 25 }
```

db.collection.mapReduce()

db.collection.**mapReduce**

The db.collection.mapReduce() provides a wrapper around the mapReduce *database command*. Always call the db.collection.mapReduce() method on a collection. The following argument list specifies a *document* with 3 required and 8 optional fields:

Parameters

- **map** – A JavaScript function that performs the "map" step of the MapReduce operation. This function references the current input document and calls the emit(key,value) method to supply the value argument to the reduce function, grouped by the key argument. Map functions may call emit(), once, more than once, or not at all depending on the type of aggregation.
- **reduce** – A JavaScript function that performs the "reduce" step of the MapReduce operation. The reduce function receives a key value and an array of emitted values from the map function, and returns a single value. Because it's possible to invoke the reduce function more than once for the same key, the structure of the object returned by function must be identical to the structure of the emitted function.
- **out** – Specifies the location of the out of the reduce stage of the operation. Specify a string to write the output of the map-reduce job to a collection with that name. The map-reduce op-

eration will replace the content of the specified collection in the current database by default. See below for additional options.

- **query** (*document*) – Optional. A query object, like the query used by the `db.collection.find()` method. Use this to specify which documents should enter the map phase of the aggregation.
- **sort** – Optional. Sorts the input objects using this key. This option is useful for optimizing the job. Common uses include sorting by the emit key so that there are fewer reduces.
- **limit** – Optional. Specifies a maximum number of objects to return from the collection.
- **finalize** – Optional. Specifies an optional "finalize" function to run on a result, following the reduce stage, to modify or control the output of the `db.collection.mapReduce()` operation.
- **scope** – Optional. Place a *document* as the contents of this field, to place fields into the global javascript scope for the execution of the map-reduce command.
- **jsMode** (*Boolean*) – Optional. Specifies whether to convert intermediate data into BSON format between the mapping and reducing steps.

  If false, map-reduce execution internally converts the values emitted during the map function from JavaScript objects into BSON objects, and so must convert those BSON objects into JavaScript objects when calling the reduce function. When this argument is false, `db.collection.mapReduce()` places the *BSON* objects used for intermediate values in temporary, on-disk storage, allowing the map-reduce job to execute over arbitrarily large data sets.

  If true, map-reduce execution retains the values emitted by the map function and returned as JavaScript objects, and so does not need to do extra conversion work to call the reduce function. When this argument is true, the map-reduce job can execute faster, but can only work for result sets with less than 500K distinct key arguments to the mapper's emit function.

  The `jsMode` option defaults to true.
- **verbose** (*Boolean*) – Optional. The `verbose` option provides statistics on job execution times.

The `out` field of the `db.collection.mapReduce()`, provides a number of additional configuration options that you may use to control how MongoDB returns data from the map-reduce job. Consider the following 4 output possibilities.

Parameters

- **replace** – Optional. Specify a collection name (e.g. `{ out: { replace: collectionName } }`) where the output of the map-reduce overwrites the contents of the collection specified (i.e. `collectionName`) if there is any data in that collection. This is the default behavior if you only specify a collection name in the `out` field.
- **merge** – Optional. Specify a collection name (e.g. `{ out: { merge: collectionName } }`) where the map-reduce operation writes output to an existing collection (i.e. `collectionName`,) and only overwrites existing documents in the collection when a new document has the same key as a document that existed before the map-reduce operation began.
- **reduce** – Optional. This operation behaves like the `merge` option above, except that when an existing document has the same key as a new document, `reduce` function from the map reduce job will run on both values and MongoDB will write the result of this function to the new collection. The specification takes the form of `{ out: { reduce:`

`collectionName } }`, where `collectionName` is the name of the results collection.

- **inline** – Optional. Indicate the inline option (i.e. `{ out: { inline: 1 } }`) to perform the map reduce job in memory and return the results at the end of the function. This option is only possible when the entire result set will fit within the *maximum size of a BSON document* (page 679). When performing map-reduce jobs on secondary members of replica sets, this is the only available `out` option.
- **db** – Optional. The name of the database that you want the map-reduce operation to write its output. By default this will be the same database as the input collection.
- **sharded** – Optional. If `true`, *and* the output mode writes to a collection, and the output database has sharding enabled, the map-reduce operation will shard the results collection according to the `_id` field.
- **nonAtomic** – New in version 2.1. Optional. Specify output operation as non-atomic such that the output behaves like a normal `multi update()`. If `true`, the post processing step will not execute inside of a database lock so that partial results will be visible during processing . `nonAtomic` is valid only for `merge` and `reduce` output operations where post-processing may be a long-running operation.

See Also:

map-reduce, provides a greater overview of MongoDB's map-reduce functionality.

Also consider "*Aggregation Framework* (page 253)" for a more flexible approach to data aggregation in MongoDB, and the "Aggregation" wiki page for an over view of aggregation in MongoDB.

db.collection.reIndex()

`db.collection.`**`reIndex`**

This method drops all indexes and recreates them. This operation may be expensive for collections that have a large amount of data and/or a large number of indexes.

Call this method, which takes no arguments, on a collection object. For example:

```
db.collection.reIndex()
```

Change `collection` to the name of the collection that you want to rebuild the index.

db.collection.remove()

`db.collection.`**`remove`**

The `remove` method removes documents from a collection.

The `remove()` method can take the following parameters:

Parameters

- **query** (*document*) – Optional. Specifies the deletion criteria using `query operators`. Omit the `query` parameter or pass an empty document (e.g. `{}`) to delete all *documents* in the *collection*.
- **justOne** (*boolean*) – Optional. A boolean that limits the deletion to just one document. The default value is `false`. Set to `true` to delete only the first result.

Note: You cannot apply the `remove()` method to a *capped collection*.

Consider the following examples of the `remove` method.

- To remove all documents in a collection, call the `remove` method with no parameters:

```
db.products.remove()
```

This operation will remove all the documents from the collection `products`.

- To remove the documents that match a deletion criteria, call the `remove` method with the `query` criteria:

```
db.products.remove( { qty: { $gt: 20 } } )
```

This operation removes all the documents from the collection `products` where `qty` is greater than `20`.

- To remove the first document that match a deletion criteria, call the `remove` method with the `query` criteria and the `justOne` parameter set to `true` or `1`:

```
db.products.remove( { qty: { $gt: 20 } }, true )
```

This operation removes all the documents from the collection `products` where `qty` is greater than `20`.

Note: If the `query` argument to the `remove()` method matches multiple documents in the collection, the delete operation may interleave with other write operations to that collection. For an unsharded collection, you have the option to override this behavior with the `$atomic` isolation operator, effectively isolating the delete operation and blocking all other operations during the delete. To isolate the query, include `$atomic: 1` in the `query` parameter as in the following example:

```
db.products.remove( { qty: { $gt: 20 }, $atomic: 1 } )
```

---

db.collection.renameCollection()

db.collection.**renameCollection**

`db.collection.renameCollection()` provides a helper for the `renameCollection` *database command* in the `mongo` shell to rename existing collections.

Parameters

- **target** (*string*) – Specifies the new name of the collection. Enclose the string in quotes.
- **dropTarget** (*boolean*) – Optional. If `true`, `mongod` will drop the *target* of `renameCollection` prior to renaming the collection.

Call the `db.collection.renameCollection()` method on a collection object, to rename a collection. Specify the new name of the collection as an argument. For example:

```
db.rrecord.renameCollection("record")
```

This operation will rename the `rrecord` collection to `record`. If the target name (i.e. `record`) is the name of an existing collection, then the operation will fail.

Consider the following limitations:

- `db.collection.renameCollection()` cannot move a collection between databases. Use `renameCollection` for these rename operations.
- `db.collection.renameCollection()` cannot operation on sharded collections.

The `db.collection.renameCollection()` method operates within a collection by changing the metadata associated with a given collection.

Refer to the documentation `renameCollection` for additional warnings and messages.

> **Warning:** The `db.collection.renameCollection()` method and `renameCollection` command will invalidate open cursors which interrupts queries that are currently returning data.

db.collection.save()

`db.collection.save`

The `save()` method updates an existing document or inserts a document depending on the parameter.

The `save()` method takes the following parameter:

Parameters

- **document** – Specify a document to save to the collection.

  If the document does not contain an *_id* field, then the `save()` method performs an insert with the specified fields in the document as well as an `_id` field with a unique *objectid* value.

  If the document contains an `_id` field, then the `save()` method performs an `upsert` querying the collection on the `_id` field:

  – If a document does not exist with the specified `_id` value, the `save()` method performs an insert with the specified fields in the document.

  – If a document exists with the specified `_id` value, the `save()` method performs an update, replacing all field in the existing record with the fields from the document.

Consider the following examples of the `save()` method:

•Pass to the `save()` method a document without an `_id` field, so that to insert the document into the collection and have MongoDB generate the unique `_id` as in the following:

```
db.products.save( { item: "book", qty: 40 } )
```

This operation inserts a new document into the `products` collection with the `item` field set to `book`, the `qty` field set to `40`, and the `_id` field set to a unique `ObjectId`:

```
{ "_id" : ObjectId("50691737d386d8fadbd6b01d"), "item" : "book", "qty" : 40 }
```

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`.

•Pass to the `save()` method a document with an `_id` field that holds a value that does not exist in the collection to insert the document with that value in the `_id` value into the collection, as in the following:

```
db.products.save( { _id: 100, item: "water", qty: 30 } )
```

This operation inserts a new document into the `products` collection with the `_id` field set to `100`, the `item` field set to `water`, and the field `qty` set to `30`:

```
{ "_id" : 100, "item" : "water", "qty" : 30 }
```

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`.

- Pass to the `save()` method a document with the `_id` field set to a value in the `collection` to replace all fields and values of the matching document with the new fields and values, as in the following:

  ```
  db.products.save( { _id:100, item:"juice" } )
  ```

  This operation replaces the existing document with a value of `100` in the `_id` field. The updated document will resemble the following:

  ```
  { "_id" : 100, "item" : "juice" }
  ```

db.collection.stats()

`db.collection.`**`stats`**

Parameters

- **scale** – Optional. Specifies the scale to deliver results. Unless specified, this command returns all sizes in bytes.

Returns A *document* containing statistics that reflecting the state of the specified collection.

This function provides a wrapper around the database command `collStats`. The `scale` option allows you to configure how the `mongo` shell scales the the sizes of things in the output. For example, specify a `scale` value of `1024` to display kilobytes rather than bytes.

Call the `db.collection.stats()` method on a collection object, to return statistics regarding that collection. For example, the following operation returns stats on the `people` collection:

```
db.people.stats()
```

See Also:

"*Collection Statistics Reference* (page 653)" for an overview of the output of this command.

db.collection.storageSize()

`db.collection.`**`storageSize`**

Returns The amount of storage space, calculated using the number of extents, used by the collection. This method provides a wrapper around the `storageSize` (page 652) output of the `collStats` (i.e. `db.collection.stats()`) command.

db.collection.totalIndexSize()

`db.collection.`**`totalIndexSize`**

Returns The total size of all indexes for the collection. This method provides a wrapper around the `db.collection.totalIndexSize()` output of the `collStats` (i.e. `db.collection.stats()`) command.

db.collection.update()

`db.collection.`**`update`**

The `update()` method modifies an existing document or documents in a collection. By default the `update()` method updates a single document. To update all documents in the collection that match the update query criteria, specify the `multi` option. To insert a document if no document matches the update query criteria,

specify the `upsert` option. Changed in version 2.2: The `mongo` shell provides an updated interface that accepts the options parameter in a document format to specify `multi` and `upsert` options. Prior to version 2.2, in the `mongo` shell, `upsert` and `multi` were positional boolean options:

```
db.collection.update(query, update, <upsert,> <multi>)
```

The `update()` method takes the following parameters:

Parameters

- **query** (*document*) – Specifies the selection criteria for the update. The `query` parameter employs the same *query selectors* as used in the `db.collection.find()` method.
- **update** (*document*) – Specifies the modifications to apply.

  If the `update` parameter contains any *update operators* expressions such as the `$set` operator expression, then:

  – the `update` parameter must contain only `update operators` expressions.

  – the `update()` method updates only the corresponding fields in the document.

  If the `update` parameter consists only of `field: value` expressions, then:

  – the `update()` method *replaces* the document with the `updates` document. If the `updates` document is missing the *_id* field, MongoDB will add the `_id` field and assign to it a unique *objectid* .

  – the `update()` method updates cannot update multiple documents.
- **options** (*document*) – New in version 2.2. Optional. Specifies whether to perform an *upsert* and/or a multiple update. Use the `options` parameter instead of the individual `upsert` and `multi` parameters.
- **upsert** (*boolean*) – Optional. Specifies an *upsert* operation

  The default value is `false`. When `true`, the `update()` method will update an existing document that matches the `query` selection criteria **or** if no document matches the criteria, insert a new document with the fields and values of the `update` parameter and if the `update` included only `update operators`, the `query` parameter as well .

  In version 2.2 of the `mongo` shell, you may also specify `upsert` in the `options` parameter.

  Note: An upsert operation affects only *one* document, and cannot update multiple documents.

- **multi** (*boolean*) – Optional. Specifies whether to update multiple documents that meet the query criteria.

  When not specified, the default value is `false` and the `update()` method updates a single document that meet the `query` criteria.

  When `true`, the `update()` method updates all documents that meet the `query` criteria.

  In version 2.2 of the `mongo` shell, you may also specify `multi` in the `options` parameter.

Although the update operation may apply mostly to updating the values of the fields, the `update()` method can also modify the name of the `field` in a document using the `$rename` operator.

Consider the following examples of the `update()` method. These examples all use the 2.2 interface to specify options in the document form.

- To update specific fields in a document, call the `update()` method with an `update` parameter using `field: value` pairs and expressions using *update operators* as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6 }, $inc: { y: 5} } )
```

This operation updates a document in the `products` collection that matches the query criteria and sets the value of the field `x` to `6`, and increment the value of the field `y` by `5`. All other fields of the document remain the same.

- To replace all the fields in a document with the document as specified in the `update` parameter, call the `update()` method with an `update` parameter that consists of *only* `key: value` expressions, as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { x: 6, y: 15 } )
```

This operation selects a document from the `products` collection that matches the query criteria sets the value of the field `x` to `6` and the value of the field `y` to `15`. All other fields of the matched document are *removed*, except the *_id* field.

- To update multiple documents, call the `update()` method and specify the `multi` option in the `options` argument, as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, { multi: t
```

This operation updates *all* documents in the `products` collection that match the query criteria by setting the value of the field `x` to `6` and the value of the field `y` to `15`. This operation does not affect any other fields in documents in the `products` collection.

You can perform the same operation by calling the `update()` method with the `multi` parameter:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, false, tru
```

- To update a document or to insert a new document if no document matches the query criteria, call the `update()` and specify the `upsert` option in the the `options` argument, as in the following:

```
db.products.update( { item: "magazine", qty: { $gt: 5 } }, { $set: { x: 25, y: 50 } }, { ups
```

This operation, will:

  - update a single document in the `products` collection that matches the query criteria, setting the value of the field `x` to `25` and the value of the field `y` to `50`, *or*
  - if no matching document exists, insert a document in the `products` collection, with the field `item` set to `magazine`, the field `x` set to `25`, and the field `y` set to `50`.

db.collection.validate()

`db.collection.`**`validate`**

Parameters

- **full** (*Boolean*) – Optional. Specify `true` to enable a full validation. MongoDB disables full validation by default because it is a potentially resource intensive operation.

Provides a wrapper around the `validate` *database command*. Call the `db.collection.validate()` method on a collection object, to validate the collection itself. Specify the full option to return full statistics.

The `validation` operation scans all of the data structures for correctness and returns a single *document* that describes the relationship between the logical collection and the physical representation of that data.

The output can provide a more in depth view of how the collection uses storage. Be aware that this command is potentially resource intensive, and may impact the performance of your MongoDB instance.

See Also:

"*Collection Validation Data* (page 655)"

db.commandHelp()

`db.commandHelp`

Parameters

- **command** – Specifies a `database command name`.

Returns Help text for the specified *database command*. See the `database command reference` for full documentation of these commands.

db.copyDatabase()

`db.copyDatabase`

Parameters

- **origin** (*database*) – Specifies the name of the database on the origin system.
- **destination** (*database*) – Specifies the name of the database that you wish to copy the origin database into.
- **hostname** (*origin*) – Indicate the hostname of the origin database host. Omit the hostname to copy from one name to another on the same server.

Use this function to copy a specific database, named `origin` running on the system accessible via `hostname` into the local database named `destination`. The command creates destination databases implicitly when they do not exist. If you omit the hostname, MongoDB will copy data from one database into another on the same instance.

This function provides a wrapper around the MongoDB *database command* "`copydb`." The `clone` database command provides related functionality.

db.createCollection()

`db.createCollection`

Parameters

- **name** (*string*) – Specifies the name of a collection to create.
- **capped** (*boolean*) – Optional. If this *document* is present, this command creates a capped collection. The capped argument is a *document* that contains the following three fields:
- **capped** – Enables a *collection cap*. False by default. If enabled, you must specify a `size` parameter.
- **size** (*bytes*) – If `capped` is `true`, `size` Specifies a maximum size in bytes, for the as a "*cap* for the collection. When `capped` is false, you may use `size`
- **max** (*int*) – Optional. Specifies a maximum "cap," in number of documents for capped collections. You must also specify `size` when specifying `max`.

Options

- **autoIndexId** – If `capped` is `true` you can specify `false` to disable the automatic index created on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 698) for more information.

Explicitly creates a new collation. Because MongoDB creates collections implicitly when referenced, this command is primarily used for creating new capped collections. In some circumstances, you may use this command to pre-allocate space for an ordinary collection.

Capped collections have maximum size or document counts that prevent them from growing beyond maximum thresholds. All capped collections must specify a maximum size, but may also specify a maximum document count. The collection will remove older documents if a collection reaches the maximum size limit before it reaches the maximum document count. Consider the following example:

```
db.createCollection("log", { capped : true, size : 536870912, max : 5000 } )
```

This command creates a collection named log with a maximum size of 5 megabytes (512 kilobytes) or a maximum of 5000 documents.

The following command simply pre-allocates a 2 gigabyte, uncapped, collection named `people`:

```
db.createCollection("people", { size: 2147483648 })
```

This command provides a wrapper around the database command `create`. See the "Capped Collections" wiki page for more information about capped collections.

db.currentOp()

`db.currentOp`

Returns A *document* that contains an array named `inprog`.

The `inprog` array reports the current operation in progress for the database instance. See *Current Operation Reporting* (page 668) for full documentation of the output of `db.currentOp()`.

`db.currentOp()` is only available for users with administrative privileges.

Consider the following JavaScript operations for the `mongo` shell that you can use to filter the output of identify specific types of operations:

•Return all pending write operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.waitingForLock && d.lockType != "read")
        printjson(d)
     })
```

•Return the active write operation:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "write")
        printjson(d)
     })
```

•Return all active read operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "read")
```

```
    printjson(d)
})
```

db.dropDatabase()

db.**dropDatabase**

Removes the current database. Does not change the current database, so the insertion of any documents in this database will allocate a fresh set of data files.

db.eval()

db.**eval**

Parameters

- **function** (*JavaScript*) – A JavaScript function.
- **arguments** – A list of arguments to pass to the JavaScript function.

Provides the ability to run JavaScript code using the JavaScript engine embedded in the MongoDB instance. In this environment the value of the db variable on the server is the name of the current database.

Unless you use db.eval(), the mongo shell itself will evaluate all JavaScript entered into mongo shell itself.

Warning: Do not use db.eval() for long running operations, as db.eval() blocks all other operations. Consider using *map-reduce* for similar functionality in these situations.
The db.eval() method cannot operate on sharded data. However, you may use db.eval() with non-sharded collections and databases stored in *sharded cluster*.

db.fsyncLock()

db.**fsyncLock**

Forces the database to flush all write operations to the disk and locks the database to prevent additional writes until the user releases the lock with the db.fsyncUnlock() command. db.fsyncLock() is an administrative command.

This command provides a simple wrapper around a fsync database command with the following syntax:

```
{ fsync: 1, lock: true }
```

This function locks the database and create a window for *backup operations* (page 180).

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)
```

db.fsyncUnlock()

db.**fsyncUnlock**

Unlocks a database server to allow writes and reverses the operation of a db.fsyncLock() operation. Typically you will use db.fsyncUnlock() following a database *backup operation* (page 180).

`db.fsyncUnlock()` is an administrative command.

db.getCollection()

`db.getCollection`

Parameters

- **name** – The name of a collection.

Returns A collection.

Use this command to obtain a handle on a collection whose name might interact with the shell itself, including collections with names that begin with _ or mirror the `database commands`.

db.getCollectionNames()

`db.getCollectionNames`

Returns An array containing all collections in the existing database.

db.getLastError()

`db.getLastError`

Returns The last error message string.

In many situation MongoDB drivers and users will follow a write operation with this command in order to ensure that the write succeeded. Use "safe mode" for most write operations.

See Also:

"*Replica Set Write Concern* (page 54)" and "`getLastError`."

db.getLastErrorObj()

`db.getLastErrorObj`

Returns A full *document* with status information.

db.getMongo()

`db.getMongo`

Returns The current database connection.

`db.getMongo()` runs when the shell initiates. Use this command to test that the `mongo` shell has a connection to the proper database instance.

db.getName()

`db.getName`

Returns the current database name.

db.getPrevError()

db.**getPrevError**

Returns A status document, containing the errors.

Deprecated since version 1.6. This output reports all errors since the last time the database received a `resetError` (also `db.resetError()`) command.

This method provides a wrapper around the `getPrevError` command.

db.getProfilingLevel()

db.**getProfilingLevel**

This method provides a wrapper around the database command "`profile`" and returns the current profiling level. Deprecated since version 1.8.4: Use `db.getProfilingStatus()` for related functionality.

db.getProfilingStatus()

db.**getProfilingStatus**

Returns The current `profile` level and `slowms` (page 627) setting.

db.getReplicationInfo()

db.**getReplicationInfo**

Returns A status document.

The output reports statistics related to replication.

See Also:

"*Replication Info Reference* (page 667)" for full documentation of this output.

db.getSiblingDB()

db.**getSiblingDB**

Used to return another database without modifying the `db` variable in the shell environment.

db.isMaster()

db.**isMaster**

Returns a status document with fields that includes the `ismaster` field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* `isMaster`

db.killOp()

db.**killOp**

Parameters

- **opid** – Specify an operation ID.

Terminates the specified operation. Use `db.currentOp()` to find operations and their corresponding ids. See *Current Operation Reporting* (page 668) for full documentation of the output of `db.currentOp()`.

db.listCommands()

`db.listCommands`
Provides a list of all database commands. See the "`http://docs.mongodb.org/manual/reference/commands`" document for a more extensive index of these options.

db.loadServerScripts()

`db.loadServerScripts`
`db.loadServerScripts()` loads all scripts in the `system.js` collection for the current database into the `mongo` shell session.

Documents in the `system.js` collection have the following prototype form:

```
{ _id : "<name>" , value : <function> } }
```

The documents in the `system.js` collection provide functions that your applications can use in any JavaScript context with MongoDB in this database. These contexts include `$where` clauses and `mapReduce` operations.

db.logout()

`db.logout`
Ends the current authentication session. This function has no effect if the current session is not authenticated.

This function provides a wrapper around the database command "`logout`".

db.printCollectionStats()

`db.printCollectionStats`
Provides a wrapper around the `db.collection.stats()` method. Returns statistics from every collection separated by three hyphen characters.

See Also:

"*Collection Statistics Reference* (page 653)"

db.printReplicationInfo()

`db.printReplicationInfo`
Provides a formatted report of the status of a *replica set* from the perspective of the *primary* set member. See the "*Replica Set Status Reference* (page 659)" for more information regarding the contents of this output.

This function will return `db.printSlaveReplicationInfo()` if issued against a *secondary* set member.

db.printShardingStatus()

`db.printShardingStatus`
Provides a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*.

See Also:

`sh.status()`

db.printSlaveReplicationInfo()

`db.printSlaveReplicationInfo`

Provides a formatted report of the status of a *replica set* from the perspective of the *secondary* set member. See the "*Replica Set Status Reference* (page 659)" for more information regarding the contents of this output.

db.removeUser()

`db.removeUser`

Parameters

- **username** – Specify a database username.

Removes the specified username from the database.

db.repairDatabase()

`db.repairDatabase`

Warning: In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` or related options like `db.repairDatabase()` in the `mongo` shell or *`mongod --repair`* (page 585). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase`. In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

`db.repairDatabase()` provides a wrapper around the database command `repairDatabase`, and has the same effect as the run-time option *`mongod --repair`* (page 585) option, limited to *only* the current database. See `repairDatabase` for full documentation.

db.resetError()

`db.resetError`

Deprecated since version 1.6. Resets the error message returned by `db.getPrevError` or `getPrevError`. Provides a wrapper around the `resetError` command.

db.runCommand()

`db.runCommand`

Parameters

- **command** (*string*) – Specifies a *database command* in the form of a *document*.
- **command** – When specifying a `command` as a string, `db.runCommand()` transforms the command into the form `{ command: 1 }`.

Provides a helper to run specified `database commands`. This is the preferred method to issue database commands, as it provides a consistent interface between the shell and drivers.

db.serverStatus()

`db.serverStatus`
Returns a *document* that provides an overview of the database process's state.

This command provides a wrapper around the database command `serverStatus`.

See Also:

"*Server Status Reference* (page 637)" for complete documentation of the output of this function.

db.setProfilingLevel()

`db.setProfilingLevel`

Parameters

- **level** – Specifies a profiling level, see list of possible values below.
- **slowms** – Optionally modify the threshold for the profile to consider a query or operation "slow."

Modifies the current *database profiler* level. This allows administrators to capture data regarding performance. The database profiling system can impact performance and can allow the server to write the contents of queries to the log, which might information security implications for your deployment.

The following profiling levels are available:

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

Also configure the `slowms` (page 627) option to set the threshold for the profiler to consider a query "slow." Specify this value in milliseconds to override the default.

This command provides a wrapper around the *database command* `profile`.

`mongod` writes the output of the database profiler to the `system.profile` collection.

`mongod` prints information about queries that take longer than the `slowms` (page 627) to the log even when the database profiler is not active.

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)
```

db.shutdownServer()

`db.shutdownServer`
Shuts down the current `mongod` or `mongos` (page 676) process cleanly and safely.

This operation fails when the current database *is not* the *admin database*.

This command provides a wrapper around the `shutdown`.

db.stats()

`db.stats`

Parameters

- **scale** – Optional. Specifies the scale to deliver results. Unless specified, this command returns all data in bytes.

Returns A *document* that contains statistics reflecting the database system's state.

This function provides a wrapper around the database command "`dbStats`". The `scale` option allows you to configure how the `mongo` shell scales the the sizes of things in the output. For example, specify a `scale` value of `1024` to display kilobytes rather than bytes.

See the "*Database Statistics Reference* (page 651)" document for an overview of this output.

Note: The scale factor rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

---

db.version()

`db.version`

Returns The version of the `mongod` instance.

fuzzFile()

`fuzzFile`

Parameters

- **filename** (*string*) – Specify a filename or path to a local file.

Returns null

For internal use.

db.getDB()

`getDB`

Returns the name of the current database as a string.

getHostName()

`getHostName`

Returns The hostname of the system running the `mongo` shell process.

getMemInfo

getMemInfo
Returns a document with two fields that report the amount of memory used by the JavaScript shell process. The fields returned are *resident* and *virtual*.

getShardDistribution()

getShardDistribution
See SERVER-4902 for more information.

getShardVersion()

getShardVersion
This method returns information regarding the state of data in a *sharded cluster* that is useful when diagnosing underlying issues with a sharded cluster.

For internal and diagnostic use only.

hostname()

hostname

Returns The hostname of the system running the `mongo` shell process.

isWindows()

_isWindows

Returns boolean.

Returns "true" if the server is running on a system that is Windows, or "false" if the server is running on a Unix or Linux systems.

listFiles()

listFiles
Returns an array, containing one document per object in the directory. This function operates in the context of the `mongo` process. The included fields are:

name
Returns a string which contains the name of the object.

isDirectory
Returns true or false if the object is a directory.

size
Returns the size of the object in bytes. This field is only present for files.

load()

load

Para string file Specify a path and file name containing JavaScript.

This native function loads and runs a JavaScript file into the current shell environment. To run JavaScript with the mongo shell, you can either:

- use the "`--eval` (page 592)" option when invoking the shell to evaluate a small amount of JavaScript code, or
- specify a file name with "*mongo* (page 592)". `mongo` will execute the script and then exit. Add the `--shell` (page 591) option to return to the shell after running the command.

Specify files loaded with the `load()` function in relative terms to the current directory of the `mongo` shell session. Check the current directory using the "`pwd()`" function.

ls()

ls

Returns a list of the files in the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

md5sumFile()

md5sumFile

Parameters

- **filename** (*string*) – a file name.

Returns The *md5* hash of the specified file.

Note: The specified filename must refer to a file located on the system running the `mongo` shell.

---

mkdir()

mkdir

Parameters

- **path** (*string*) – A path on the local filesystem.

Creates a directory at the specified path. This command will create the entire path specified, if the enclosing directory or directories do not already exit.

Equivalent to **mkdir -p** with BSD or GNU utilities.

mongo.setSlaveOk()

`mongo.`**setSlaveOk**

For the current session, this command permits read operations from non-master (i.e. *slave* or *secondary*) instances. Practically, use this method in the following form:

```
db.getMongo().setSlaveOk()
```

Indicates that "*eventually consistent*" read operations are acceptable for the current application. This function provides the same functionality as `rs.slaveOk()`.

See the `readPref()` method for more fine-grained control over *read preference* (page 55) in the `mongo` shell.

pwd()

pwd

Returns the current directory.

This function returns with output relative to the current shell session, and does not impact the server.

quit()

quit

Exits the current shell session.

rand()

_rand

Returns A random number between `0` and `1`.

This function provides functionality similar to the `Math.rand()` function from the standard library.

rawMongoProgramOutput()

rawMongoProgramOutput

For internal use.

removeFile()

removeFile

Parameters

- **filename** (*string*) – Specify a filename or path to a local file.

Returns boolean.

Removes the specified file from the local file system.

resetDbpath()

resetDbpath

For internal use.

rs.add()

`rs.add`

Specify one of the following forms:

Parameters

- **host** (*string*) – Either a string or a document. If a string, specifies a host (and optionally port-number) for a new host member for the replica set; MongoDB will add this host with the default configuration. If a document, specifies any attributes about a member of a replica set.
- **arbiterOnly** (*Boolean*) – Optional. If `true`, this host is an arbiter. If the second argument evaluates to `true`, as is the case with some *documents*, then this instance will become an arbiter.

Provides a simple method to add a member to an existing *replica set*. You can 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 78) example.
2. as a configuration *document*, as in the *Add a Member to an Existing Replica Set (Alternate Procedure)* (page 79) example.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node 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 661) document for full documentation of all replica set configuration options.

rs.addArb()

`rs.addArb`

Parameters

- **host** (*string*) – Specifies a host (and optionally port-number) for a arbiter member for the replica set.

Adds a new *arbiter* to an existing replica set.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

rs.conf()

`rs.conf`

Returns a *document* that contains the current *replica set* configuration object.

`rs.config`

`rs.config()` is an alias of `rs.conf()`.

rs.freeze()

`rs.freeze`

Parameters

- **seconds** (*init*) – Specify the duration of this operation.

Forces the current node to become ineligible to become primary for the period specified.

`rs.freeze()` provides a wrapper around the *database command* `replSetFreeze`.

rs.help()

`rs.help`
Returns a basic help text for all of the *replication* (page 33) related shell functions.

rs.initiate()

`rs.initiate`

Parameters

- **configuration** – Optional. A *document* that specifies the configuration of a replica set. If not specified, MongoDB will use a default configuration.

Initiates a replica set. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set. Consider the following model of the most basic configuration for a 3-member replica set:

```
{
    _id : <setname>,
     members : [
         {_id : 0, host : <host0>},
         {_id : 1, host : <host1>},
         {_id : 2, host : <host2>},
     ]
}
```

This function provides a wrapper around the "`replSetInitiate`" *database command*.

rs.reconfig()

`rs.reconfig`

Parameters

- **configuration** – A *document* that specifies the configuration of a replica set.
- **force** – Optional. Specify `{ force: true }` as the force parameter to force 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 *rollback* situations.

Initializes a new *replica set* configuration. This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.reconfig()` provides a wrapper around the "`replSetReconfig`" *database command*.

`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.

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 isn't connected to the current member, or you're 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 661)" and "*Replica Set Administration* (page 38)".

rs.remove()

`rs.remove`

Parameters

- **hostname** – Specify one of the existing hosts to remove from the current replica set.

Removes the node 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 negotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

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()

`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* nodes. See the `readPref()` method for more fine-grained control over *read preference* (page 55) in the `mongo` shell.

rs.status()

`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*.

See Also:

"*Replica Set Status Reference* (page 659)" for documentation of this output.

rs.stepDown()

`rs.`**`stepDown`**

Parameters

- **seconds** (*init*) – Specify the duration of this operation. If not specified the command uses the default value of 60 seconds.

Returns disconnects shell.

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 node is not primary.

This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which node 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` (page 519).

rs.syncFrom()

`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 sync from in the form of `[hostname]:[port]`.

See `replSetSyncFrom` for more details.

run()

`run`

For internal use.

runMongoProgram()

`runMongoProgram`

For internal use.

runProgram()

`runProgram`

For internal use.

sh.addShard()

sh.**addShard**

Parameters

- **host** – Specify the hostname of a new shard server.

Use this to add shard instances to the current *cluster*. The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[set]/[hostname]
[set]/[hostname],[hostname]:port
```

You can specify shards using the hostname, or a hostname and port combination if the shard is running on a non-standard port. A *replica set* can also function as a shard member. In these cases supply `addShard` with the set name, followed by at least one existing member of the set as a seed in a comma separated list, as in the final two examples.

This function provides a wrapper around the administrative command `addShard`.

sh.addShardTag()

sh.**addShardTag**

New in version 2.2.

Parameters

- **shard** – Specifies the name of the shard that you want to give a specific tag.
- **tag** – Specifies the name of the tag that you want to add to the shard.

`sh.addShardTag()` associates a shard with a tag or identifier. MongoDB can use these identifiers, to "home" or attach (i.e. with `sh.addTagRange()`) specific data to a specific shard.

Always issue `sh.addShardTag()` when connected to a `mongos` (page 676) instance. The following example adds three tags, `LGA`, `EWR`, and `JFK`, to three shards:

```
sh.addShardTag("shard0000", "LGA")
sh.addShardTag("shard0001", "EWR")
sh.addShardTag("shard0002", "JFK")
```

sh.addTagRange()

sh.**addTagRange**

New in version 2.2.

Parameters

- **namespace** – Specifies the namespace, in the form of `<database>.<collection>` of the sharded collection that you would like to tag.
- **minimum** – Specifies the minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`.
- **maximum** – Specifies the maximum value of the shard key range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`.
- **tag** – Specifies the name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

`sh.addTagRange()` attaches a range of values of the shard key to a shard tag created using the `sh.addShardTag()` helper. Use this operation to ensure that the documents that exist within the specified range exist on shards that have a matching tag.

Always issue `sh.addTagRange()` when connected to a `mongos` (page 676) instance.

sh.enableSharding()

`sh.`**`enableSharding`**

Parameters

- **database** (*name*) – Specify a database name to shard.

Enables sharding on the specified database. This does not automatically shard any collections, but makes it possible to begin sharding collections using `sh.shardCollection()`.

sh.getBalancerState()

sh.help()

`sh.`**`help`**

Returns a basic help text for all sharding related shell functions.

sh.isBalancerRunning()

`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.

sh.moveChunk()

`sh.`**`moveChunk`**

Parameters

- **collection** (*string*) – Specify the sharded collection containing the chunk to migrate.
- **query** – Specify a query to identify documents in a specific chunk. Typically specify the *shard key* for a document as the query.
- **destination** (*string*) – Specify the name of the shard that you wish to move the designated chunk to.

Moves the chunk containing the documents specified by the `query` to the shard described by `destination`.

This function provides a wrapper around the `moveChunk`. In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` directly.

See Also:

"`moveChunk`" and "*Sharding* (page 105)" for more information.

sh.removeShardTag()

`sh.removeShardTag`
New in version 2.2.

Parameters

- **shard** – Specifies the name of the shard that you want to remove a tag from.
- **tag** – Specifies the name of the tag that you want to remove from the shard.

Removes the association between a tag and a shard. Always issue `sh.removeShardTag()` when connected to a `mongos` (page 676) instance.

sh.setBalancerState()

`sh.setBalancerState`

Parameters

- **state** (*boolean*) – `true` enables the balancer if disabled, and `false` disables the balancer.

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.

`sh.getBalancerState`

Returns boolean.

`sh.getBalancerState()` returns `true` when the *balancer* is enabled and false when the balancer is disabled. This does not reflect the current state of balancing operations: use `sh.isBalancerRunning()` to check the balancer's current state.

sh.shardCollection()

`sh.shardCollection`

Parameters

- **collection** (*name*) – The name of the collection to shard.
- **key** (*document*) – A *document* containing *shard key* that the sharding system uses to *partition* and distribute objects among the shards.
- **unique** (*boolean*) – When true, the `unique` option ensures that the underlying index enforces uniqueness so long as the unique index is a prefix of the shard key.

Shards the named collection, according to the specified *shard key*. Specify shard keys in the form of a *document*. Shard keys may refer to a single document field, or more typically several document fields to form a "compound shard key."

sh.splitAt()

`sh.splitAt`

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.

- **query** (*document*) – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the 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. 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()`.

sh.splitFind()

`sh.`**`splitFind`**

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.
- **query** – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the 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()`.

sh.status()

`sh.`**`status`**

Returns a formatted report of the status of the *sharded cluster*, including data regarding the distribution of chunks.

srand()

`_srand`

For internal use.

startMongoProgram()

`_startMongoProgram`

For internal use.

stopMongoProgram()

`stopMongoProgram`

For internal use.

stopMongoProgramByPid()

stopMongoProgramByPid
For internal use.

stopMongod()

stopMongod
For internal use.

waitMongoProgramOnPort()

waitMongoProgramOnPort
For internal use.

waitProgram()

waitProgram
For internal use.

36.2 MongoDB and SQL Interface Comparisons

The following documents provide mappings between MongoDB concepts and statements and SQL concepts and statements.

36.2.1 SQL to MongoDB Mapping Chart

Executables

The following table presents the MySQL/Oracle executables and the corresponding MongoDB executables.

	MySQL/Oracle	MongoDB
Database Server	mysqld/oracle	*mongod* (page 581)
Database Client	mysql/sqlplus	*mongo* (page 591)

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
database	*database*
table	*collection*
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key Specify any unique column or column combination as primary key.	*primary key* In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation framework See the *SQL to Aggregation Framework Mapping Chart* (page 576).

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
`CREATE TABLE users ( id MEDIUMINT NOT NULL AUTO_INCREMENT, user_id Varchar(30), age Number, status char(1), PRIMARY KEY (id) )`	Implicitly created on first `insert` operation. The primary key `_id` is automatically added if `_id` field is not specified. `db.users.insert( { user_id: "abc123", age: 55, status: "A" } )` However, you can also explicitly create a collection: `db.createCollection("users")`	See `insert()` and `createCollection()` (page 552) for more information.
`ALTER TABLE users ADD join_date DATETIME`	Collections do not describe or enforce the structure of the constituent documents. See the Schema Design wiki page for more information.	See `update()` and `$set` for more information on changing the structure of documents in a collection.
`ALTER TABLE users DROP COLUMN join_date`	Collections do not describe or enforce the structure of the constituent documents. See the Schema Design wiki page for more information.	See `update()` and `$set` for more information on changing the structure of documents in a collection.
`CREATE INDEX idx_user_id_asc ON users(user_id)`	`db.users.ensureIndex( { user_id: 1 } )`	See `ensureIndex()` and *indexes* (page 231) for more information.
`CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)`	`db.users.ensureIndex( { user_id: 1, age: -1 } )`	See `ensureIndex()` and *indexes* (page 231) for more information.
`DROP TABLE users`	`db.users.drop()`	See `drop()` 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
`INSERT INTO users(user_id, age, status) VALUES ("bcd001", 45, "A")`	`db.users.insert( { user_id: "bcd001", age: 45, status: "A" } )`	See `insert()` 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
`SELECT * FROM users`	`db.users.find()`	See `find()` for more information.
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`	See `find()` for more information.
`SELECT user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1, _id: 0 } )`	See `find()` for more information.
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`	See `find()` for more information.
`SELECT user_id, status FROM users WHERE status = "A"`	`db.users.find( { status: "A" }, { user_id: 1, status: 1, _id: 0 } )`	See `find()` for more information.
`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`	See `find()` and `$ne` for more information.
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`	See `find()` and `$and` for more information.
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`	See `find()` and `$or` for more information.
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`	See `find()` and `$gt` for more information.
`SELECT * FROM users WHERE age < 25`	`db.users.find( { age: { $lt: 25 } } )`	See `find()` and `$lt` for more information.
`SELECT * FROM users WHERE age > 25 AND age <= 50`	`db.users.find( { age: { $gt: 25, $lte: 50 } } )`	See `find()`, `$gt`, and `$lte` for more information.
`SELECT * FROM users WHERE user_id like "%bc%"`	`db.users.find( { user_id: /bc/ } )`	See `find()` and `$regex` 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
`UPDATE users SET status = "C" WHERE age > 25`	`db.users.update( { age: { $gt: 25 } }, { $set: { status: "C" } }, { multi: true } )`	See `update()`, `$gt`, and `$set` for more information.
`UPDATE users SET age = age + 3 WHERE status = "A"`	`db.users.update( { status: "A" } , { $inc: { age: 3 } }, { multi: true } )`	See `update()`, `$inc`, and `$set` 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
`DELETE FROM users WHERE status = "D"`	`db.users.remove( { status: "D" } )`	See `remove()` for more information.
`DELETE FROM users`	`db.users.remove( )`	See `remove()` for more information.

36.2.2 SQL to Aggregation Framework Mapping Chart

The *aggregation framework* (page 253) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators* (page 270):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 475)
GROUP BY	`$group` (page 473)
HAVING	`$match` (page 475)
SELECT	`$project` (page 477)
ORDER BY	`$sort` (page 479)
LIMIT	`$limit` (page 475)
SUM()	`$sum` (page 481)
COUNT()	`$sum` (page 481)
join	No direct corresponding operator; *however*, the `$unwind` (page 481) 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
`SELECT COUNT(*) AS count FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, count: { $sum: 1 } } } ] )`	Count all records from `orders`
`SELECT SUM(price) AS total FROM orders`	`db.orders.aggregate( [ { $group: { _id: null, total: { $sum: "$price" } } } ] )`	Sum the `price` field from `orders`
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, sum the `price` field.
`SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", total: { $sum: "$price" } } }, { $sort: { total: 1 } } ] )`	For each unique `cust_id`, sum the `price` field, results sorted by sum.
`SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date`	`db.orders.aggregate( [ { $group: { _id: { cust_id: "$cust_id", ord_date: "$ord_date" }, total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, `ord_date` grouping, sum the `price` field.
`SELECT cust_id, count(*) FROM orders GROUP BY cust_id HAVING count(*) > 1`	`db.orders.aggregate( [ { $group: { _id: "$cust_id", count: { $sum: 1 } } }, { $match: { count: { $gt: 1 } } } ] )`	For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.
`SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date HAVING total > 250`	`db.orders.aggregate( [ { $group: { _id: { cust_id: "$cust_id", ord_date: "$ord_date" }, total: { $sum: "$price" } } }, { $match: { total: { $gt: 250 } } } ] )`	For each unique `cust_id`, `ord_date` grouping, sum the `price` field and return only where the sum is greater than 250.
`SELECT cust_id, SUM(price) as total FROM orders WHERE status = 'A' GROUP BY cust_id`	`db.orders.aggregate( [ { $match: { status: 'A' } }, { $group: { _id: "$cust_id", total: { $sum: "$price" } } } ] )`	For each unique `cust_id` with status `A`, sum the `price` field.
`SELECT cust_id, SUM(price) as total FROM orders WHERE status = 'A' GROUP BY cust_id HAVING total > 250`	`db.orders.aggregate( [ { $match: { status: 'A' } }, { $group: { _id: "$cust_id", total: { $sum: "$price" } } }, { $match: { total: { $gt: 250 } } } ] )`	For each unique `cust_id` with status `A`, sum the `price` field and return only where the sum is greater than 250.

36.3 Overviews

For this reference material in another form, consider the following interface overview pages:

- `http://docs.mongodb.org/manual/reference/operators` for an overview of all query, update, and projection operators;
- `http://docs.mongodb.org/manual/reference/meta-query-operators` for all special "meta" query operators;
- *Aggregation Framework Reference* (page 269) for all *aggregation* (page 251) operators;
- `http://docs.mongodb.org/manual/reference/commands` for an overview of all *database commands*; and
- `http://docs.mongodb.org/manual/reference/javascript` for all `mongo` shell methods and helpers.

CHAPTER

THIRTYSEVEN

ARCHITECTURE AND COMPONENTS

37.1 MongoDB Package Components

37.1.1 Core Processes

The core components in the MongoDB package are: `mongod`, the core database process; `mongos` (page 676) the controller and query router for *sharded clusters*; and `mongo` the interactive MongoDB Shell.

mongod

Synopsis

`mongod` is the primary daemon process for the MongoDB system. It handles data requests, manages data format, and performs background management operations.

This document provides a complete overview of all command line options for `mongod`. These options are primarily useful for testing purposes. In common operation, use the *configuration file options* (page 621) to control the behavior of your database, which is fully capable of all operations described below.

Options

mongod

--help, -h
Returns a basic help and usage text.

--version
Returns the version of the `mongod` daemon.

--config `<filename>`**, -f** `<filename>`
Specifies a configuration file, that you can use to specify runtime-configurations. While the options are equivalent and accessible via the other command line arguments, the configuration file is the preferred method for runtime configuration of mongod. See the "*Configuration File Options* (page 621)" document for more information about these options.

--verbose, -v
Increases the amount of internal reporting returned on standard output or in the log file specified by `--logpath` (page 582). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

--quiet
Runs the mongod instance in a quiet mode that attempts to limit the amount of output. This option suppresses:

- output from *database commands*, including drop, dropIndex, diagLogging, validate, and clean.
- replication activity.
- connection accepted events.
- connection closed events.

--port <port>
Specifies a TCP port for the mongod to listen for client connections. By default mongod listens for connections on port 27017.

UNIX-like systems require root privileges to use ports with numbers lower than 1000.

--bind_ip <ip address>
The IP address that the mongod process will bind to and listen for connections. By default mongod listens for connections on the localhost (i.e. 127.0.0.1 address.) You may attach mongod to any interface; however, if you attach mongod to a publicly accessible interface ensure that you have implemented proper authentication and/or firewall restrictions to protect the integrity of your database.

--maxConns <number>
Specifies the maximum number of simultaneous connections that mongod will accept. This setting will have no effect if it is higher than your operating system's configured maximum connection tracking threshold.

Note: You cannot set maxConns (page 622) to a value higher than *20000*.

--objcheck
Forces the mongod to validate all requests from clients upon receipt to ensure that invalid objects are never inserted into the database. Enabling this option will produce some performance impact, and is not enabled by default.

--logpath <path>
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, mongod will output all log information to the standard output. Additionally, unless you also specify *--logappend* (page 582), the logfile will be overwritten when the process restarts.

Note: The behavior of the logging system may change in the near future in response to the SERVER-4499 case.

--logappend
When specified, this option ensures that mongod appends new entries to the end of the logfile rather than overwriting the content of the log when the process restarts.

--syslog
Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with *--logpath* (page 582).

> **Warning:** You cannot use *--syslog* (page 582) with *--logpath* (page 582).

--pidfilepath <path>
Specify a file location to hold the "*PID*" or process ID of the mongod process. Useful for tracking the mongod process in combination with the *mongod --fork* (page 583) option.

If this option is not set, mongod will create no PID file.

--keyFile <file>
Specify the path to a key file to store authentication information. This option is only useful for the connection between replica set members.

See Also:

"*Replica Set Security* (page 46)" and "*Replica Set Administration* (page 38)."

--nounixsocket
Disables listening on the UNIX socket. Unless set to false, mongod and mongos (page 676) provide a UNIX-socket.

--unixSocketPrefix <path>
Specifies a path for the UNIX socket. Unless this option has a value, mongod and mongos (page 676), create a socket with the http://docs.mongodb.org/manual/tmp as a prefix.

--fork
Enables a *daemon* mode for mongod that runs the process to the background. This is the normal mode of operation, in production and production-like environments, but may *not* be desirable for testing.

--auth
Enables database authentication for users connecting from remote hosts. configure users via the *mongo shell shell* (page 591). If no users exist, the localhost interface will continue to have access to the database until the you create the first user.

See the "Security and Authentication wiki page for more information regarding this functionality.

--cpu
Forces mongod to report the percentage of CPU time in write lock. mongod generates output every four seconds. MongoDB writes this data to standard output or the logfile if using the logpath (page 622) option.

--dbpath <path>
Specify a directory for the mongod instance to store its data. Typical locations include: http://docs.mongodb.org/manual/srv/mongodb, http://docs.mongodb.org/manual/var/lib/mongodb or http://docs.mongodb.org/manual/opt/mon

Unless specified, mongod will look for data files in the default http://docs.mongodb.org/manual/data/db directory. (Windows systems use the \data\db directory.) If you installed using a package management system. Check the http://docs.mongodb.org/manual/etc/mongodb.conf file provided by your packages to see the configuration of the dbpath (page 624).

--diaglog <value>
Creates a very verbose, *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the dbpath (page 624) directory in a series of files that begin with the string diaglog and end with the initiation time of the logging as a hex string.

The specified value configures the level of verbosity. Possible values, and their impact are as follows.

Value	Setting
0	off. No logging.
1	Log write operations.
2	Log read operations.
3	Log both read and write operations.
7	Log write and some read operations.

You can use the mongosniff tool to replay this output for investigation. Given a typical diaglog file, located at http://docs.mongodb.org/manual/data/db/diaglog.4f76a58c, you might use a command in the following form to read these files:

```
mongosniff --source DIAGLOG /data/db/diaglog.4f76a58c
```

--diaglog (page 583) is for internal use and not intended for most users.

Warning: Setting the diagnostic level to 0 will cause mongod to stop writing data to the *diagnostic log* file. However, the mongod instance will continue to keep the file open, even if it is no longer writing data to the file. If you want to rename, move, or delete the diagnostic log you must cleanly shut down the mongod instance before doing so.

--directoryperdb
Alters the storage pattern of the data directory to store each database's files in a distinct folder. This option will create directories within the *--dbpath* (page 583) named for each directory.

Use this option in conjunction with your file system and device configuration so that MongoDB will store data on a number of distinct disk devices to increase write throughput or disk capacity.

--journal
Enables operation journaling to ensure write durability and data consistency. mongodb enables journaling by default on 64-bit builds of versions after 2.0.

--journalOptions <arguments>
Provides functionality for testing. Not for general use, and may affect database integrity.

--journalCommitInterval <value>
Specifies the maximum amount of time for mongod to allow between journal operations. The default value is 100 milliseconds, while possible values range from 2 to 300 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance.

To force mongod to commit to the journal more frequently, you can specify j:true. When a write operation with j:true pending, mongod will reduce journalCommitInterval (page 625) to a third of the set value.

--ipv6
Specify this option to enable IPv6 support. This will allow clients to connect to mongod using IPv6 networks. mongod disables IPv6 support by default in mongod and all utilities.

--jsonp
Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

--noauth
Disable authentication. Currently the default. Exists for future compatibility and clarity.

--nohttpinterface
Disables the HTTP interface.

--nojournal
Disables the durability journaling. By default, mongod enables journaling in 64-bit versions after v2.0.

--noprealloc
Disables the preallocation of data files. This will shorten the start up time in some cases, but can cause significant performance penalties during normal operations.

--noscripting
Disables the scripting engine.

--notablescan
Forbids operations that require a table scan.

--nssize <value>
Specifies the default value for namespace files (i.e `.ns`). This option has no impact on the size of existing namespace files.

The default value is 16 megabytes, this provides for effectively 12,000 possible namespaces. The maximum size is 2 gigabytes.

--profile <level>
Changes the level of database profiling, which inserts information about operation performance into output of `mongod` or the log file. The following levels are available:

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

Profiling is off by default. Database profiling can impact database performance. Enable this option only after careful consideration.

--quota
Enables a maximum limit for the number data files each database can have. When running with *--quota* (page 585), there are a maximum of 8 data files per database. Adjust the quota with the *--quotaFiles* (page 585) option.

--quotaFiles <number>
Modify limit on the number of data files per database. This option requires the *--quota* (page 585) setting. The default value for *--quotaFiles* (page 585) is 8.

--rest
Enables the simple *REST* API.

--repair
Runs a repair routine on all databases. This is equivalent to shutting down and running `repairDatabase` database command on all databases.

Warning: In general, if you have an intact copy of your data, such as would exist on a very recent backup or an intact member of a *replica set*, **do not** use `repairDatabase` or related options like `db.repairDatabase()` in the `mongo` shell or *mongod --repair* (page 585). Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase`. In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

Changed in version 2.1.2. If you run the repair option *and* have data in a journal file, `mongod` will refuse to start. In these cases you should start `mongod` without the *--repair* (page 585) option to allow `mongod` to recover data from the journal. This will complete more quickly and will result in a more consistent and complete data set.

To continue the repair operation despite the journal files, shut down `mongod` cleanly and restart with the *--repair* (page 585) option.

--repairpath <path>
Specifies the root directory containing MongoDB data files, to use for the *--repair* (page 585) operation. Defaults to the value specified by *--dbpath* (page 583).

--slowms <value>
Defines the value of "slow," for the *--profile* (page 585) option. The database logs all slow queries to the

log, even when the profiler is not turned on. When the database profiler is on, mongod the profiler writes to the system.profile collection. See profile for more information on the database profiler.

--smallfiles
Enables a mode where MongoDB uses a smaller default file size. Specifically, *--smallfiles* (page 586) reduces the initial size for data files and limits them to 512 megabytes. *--smallfiles* (page 586) also reduces the size of each *journal* files from 1 gigabyte to 128 megabytes.

Use *--smallfiles* (page 586) if you have a large number of databases that each holds a small quantity of data. *--smallfiles* (page 586) can lead your mongod to create a large number of files, which may affect performance for larger databases.

--shutdown
Used in *control scripts*, the *--shutdown* (page 586) will cleanly and safely terminate the mongod process. When invoking mongod with this option you must set the *--dbpath* (page 583) option either directly or by way of the *configuration file* (page 621) and the *--config* (page 581) option.

--syncdelay <value>
This setting controls the maximum number of seconds between disk syncs. While mongod is always writing data to disk, this setting controls the maximum guaranteed interval between a successful write operation and the next time the database flushes data to disk.

In many cases, the actual interval between write operations and disk flushes is much shorter than the value

If set to 0, mongod flushes all operations to disk immediately, which may have a significant performance impact. If *--journal* (page 584) is true, all writes will be durable, by way of the journal within the time specified by *--journalCommitInterval* (page 584).

--sysinfo
Returns diagnostic system information and then exits. The information provides the page size, the number of physical pages, and the number of available physical pages.

--upgrade
Upgrades the on-disk data format of the files specified by the *--dbpath* (page 583) to the latest version, if needed.

This option only affects the operation of mongod if the data files are in an old format.

Note: In most cases you should **not** set this value, so you can exercise the most control over your upgrade process. See the MongoDB release notes (on the download page) for more information about the upgrade process.

--traceExceptions
For internal diagnostic use only.

Replication Options

--replSet <setname>
Use this option to configure replication with replica sets. Specify a setname as an argument to this set. All hosts must have the same set name.

See Also:

"*Replication* (page 31)," "*Replica Set Administration* (page 38)," and "*Replica Set Configuration* (page 661)"

--oplogSize <value>
Specifies a maximum size in megabytes for the replication operation log (e.g. *oplog*.) By mongod creates an *oplog* based on the maximum amount of space available. For 64-bit systems, the op log is typically 5% of available disk space.

Once the mongod has created the oplog for the first time, changing --oplogSize (page 586) will not affect the size of the oplog.

--fastsync
In the context of *replica set* replication, set this option if you have seeded this replica with a snapshot of the *dbpath* of another member of the set. Otherwise the mongod will attempt to perform a full sync.

> **Warning:** If the data is not perfectly synchronized *and* mongod starts with fastsync (page 628), then the secondary or slave will be permanently out of sync with the primary, which may cause significant consistency problems.

--replIndexPrefetch
New in version 2.2. You must use --replIndexPrefetch (page 587) in conjunction with replSet (page 628). The default value is all and available options are:

- none
- all
- _id_only

By default *secondary* members of a *replica set* will load all indexes related to an operation into memory before applying operations from the oplog. You can modify this behavior so that the secondaries will only load the _id index. Specify _id_only or none to prevent the mongod from loading *any* index into memory.

Master/Slave Replication These options provide access to conventional master-slave database replication. While this functionality remains accessible in MongoDB, replica sets are the preferred configuration for database replication.

--master
Configures mongod to run as a replication *master*.

--slave
Configures mongod to run as a replication *slave*.

--source <host>:<port>
For use with the --slave (page 587) option, the --source option designates the server that this instance will replicate.

--only <arg>
For use with the --slave (page 587) option, the --only option specifies only a single *database* to replicate.

--slavedelay <value>
For use with the --slave (page 587) option, the --slavedelay option configures a "delay" in seconds, for this slave to wait to apply operations from the *master* node.

--autoresync
For use with the --slave (page 587) option, the --autoresync (page 587) option allows this slave to automatically resync if the local data is more than 10 seconds behind the master. This option may be problematic if the *oplog* is too small (controlled by the --oplogSize (page 586) option.) If the *oplog* not large enough to store the difference in changes between the master's current state and the state of the slave, this node will forcibly resync itself unnecessarily. When you set the If the --autoresync (page 587) option the slave will not attempt an automatic resync more than once in a ten minute period.

Sharding Cluster Options
--configsvr
Declares that this mongod instance serves as the *config database* of a sharded cluster. When running with this option, clients will not be able to write data to any database other than config and admin. The default

port for :program:`mongod` with this option is ``27019 and mongod writes all data files to the http://docs.mongodb.org/manual/configdb sub-directory of the *--dbpath* (page 583) directory.

--shardsvr
Configures this mongod instance as a shard in a partitioned cluster. The default port for these instances is 27018. The only affect of *--shardsvr* (page 588) is to change the port number.

--noMoveParanoia
Disables a "paranoid mode" for data writes for chunk migration operation. See the *chunk migration* (page 138) and moveChunk command documentation for more information.

By default mongod will save copies of migrated chunks on the "from" server during migrations as "paranoid mode." Setting this option disables this paranoia.

Usage

In common usage, the invocation of mongod will resemble the following in the context of an initialization or control script:

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

See the "*Configuration File Options* (page 621)" for more information on how to configure mongod using the configuration file.

mongos

Synopsis

mongos (page 676) for "MongoDB Shard," is a routing service for MongoDB shard configurations that processes queries from the application layer, and determines the location of this data in the *sharded cluster*, in order to complete these operations. From the perspective of the application, a mongos (page 676) instance behaves identically to any other MongoDB instance.

See Also:

See the "Sharding" wiki page for more information regarding MongoDB's sharding functionality.

Note: Changed in version 2.1. Some aggregation operations using the aggregate will cause mongos (page 676) instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architecture decisions if you make use the *aggregation framework* extensively in a sharded environment.

Options

mongos

--help, -h
Returns a basic help and usage text.

--version
Returns the version of the mongod daemon.

--config <filename>**, -f** <filename>
Specifies a configuration file, that you can use to specify runtime-configurations. While the options are equivalent and accessible via the other command line arguments, the configuration file is the preferred method for

runtime configuration of mongod. See the "*Configuration File Options* (page 621)" document for more information about these options.

Not all configuration options for `mongod` make sense in the context of `mongos` (page 676).

`--verbose, -v`
Increases the amount of internal reporting returned on standard output or in the log file specified by `--logpath` (page 589). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

`--quiet`
Runs the `mongos` (page 676) instance in a quiet mode that attempts to limit the amount of output.

`--port` `<port>`
Specifies a TCP port for the `mongos` (page 676) to listen for client connections. By default `mongos` (page 676) listens for connections on port 27017.

UNIX-like systems require root access to access ports with numbers lower than 1000.

`--bind_ip` `<ip address>`
The IP address that the `mongos` (page 676) process will bind to and listen for connections. By default `mongos` (page 676) listens for connections on the localhost (i.e. `127.0.0.1` address.) You may attach `mongos` (page 676) to any interface; however, if you attach `mongos` (page 676) to a publicly accessible interface you must implement proper authentication or firewall restrictions to protect the integrity of your database.

`--maxConns` `<number>`
Specifies the maximum number of simultaneous connections that `mongos` (page 676) will accept. This setting will have no effect if the value of this setting is higher than your operating system's configured maximum connection tracking threshold.

This is particularly useful for `mongos` (page 676) if you have a client that creates a number of collections but allows them to timeout rather than close the collections. When you set `maxConns` (page 622), ensure the value is slightly higher than the size of the connection pool or the total number of connections to prevent erroneous connection spikes from propagating to the members of a *shard* cluster.

Note: You cannot set `maxConns` (page 622) to a value higher than *20000*.

`--objcheck`
Forces the `mongos` (page 676) to validate all requests from clients upon receipt to ensure that invalid objects are never inserted into the database. This option has a performance impact, and is not enabled by default.

`--logpath` `<path>`
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, `mongos` (page 676) will output all log information to the standard output. Additionally, unless you also specify `--logappend` (page 589), the logfile will be overwritten when the process restarts.

`--logappend`
Specify to ensure that `mongos` (page 676) appends additional logging data to the end of the logfile rather than overwriting the content of the log when the process restarts.

`--syslog`
Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with `--logpath` (page 589).

Warning: You cannot use `--syslog` (page 589) with `--logpath` (page 589).

--pidfilepath <path>
Specify a file location to hold the "*PID*" or process ID of the mongod process. Useful for tracking the mongod process in combination with the *mongos --fork* (page 590) option.

Without this option, mongos (page 676) will create a PID file.

--keyFile <file>
Specify the path to a key file to store authentication information. This option is only useful for the connection between mongos (page 676) instances and components of the *sharded cluster*.

See Also:

"*Replica Set Security* (page 46)" and "*Replica Set Administration* (page 38)."

--nounixsocket
Disables listening on the UNIX socket. Without this option mongos (page 676) creates a UNIX socket.

--unixSocketPrefix <path>
Specifies a path for the UNIX socket. Unless specified, mongos (page 676) creates a socket in the http://docs.mongodb.org/manual/tmp path.

--fork
Enables a *daemon* mode for mongod which forces the process to the background. This is the normal mode of operation, in production and production-like environments, but may *not* be desirable for testing.

--configdb <config1>,<config2><:port>,<config3>
Set this option to specify a configuration database (i.e. *config database*) for the *sharded cluster*. You must specify either 1 configuration server or 3 configuration servers, in a comma separated list.

Note: Each mongos (page 676) reads from the first *config server* in the list provided. If your configuration databases reside in more that one data center, you should specify the closest config servers as the first servers in the list.

Warning: Never remove a config server from the *--configdb* (page 590) parameter, even if the config server or servers are not available, or offline.

--test
This option is for internal testing use only, and runs unit tests without starting a mongos (page 676) instance.

--upgrade
This option updates the meta data format used by the *config database*.

--chunkSize <value>
The value of the *--chunkSize* (page 590) determines the size of each *chunk* of data distributed around thee *sharded cluster*. The default value is 64 megabytes, which is the ideal size for chunks in most deployments: larger chunk size can lead to uneven data distribution, smaller chunk size often leads to inefficient movement of chunks between nodes. However, in some circumstances it may be necessary to set a different chunk size.

This option *only* sets the chunk size when initializing the cluster for the first time. If you modify the run-time option later, the new value will have no effect. See the "*Modify Chunk Size* (page 122)" procedure if you need to change the chunk size on an existing sharded cluster.

--ipv6
Enables IPv6 support to allow clients to connect to mongos (page 676) using IPv6 networks. MongoDB disables IPv6 support by default in mongod and all utilities.

--jsonp
Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

--noscripting
Disables the scripting engine.

--nohttpinterface
New in version 2.1.2. Disables the HTTP interface.

--localThreshold
New in version 2.2. `--localThreshold` (page 591) affects the logic that program:*mongos* uses when selecting *replica set* members to pass reads operations to from clients. Specify a value to `--localThreshold` (page 591) in milliseconds. The default value is `15`, which corresponds to the default value in all of the client *drivers* (page 285).

When `mongos` (page 676) receives a request that permits reads to *secondary* members, the `mongos` (page 676) will:

- find the member of the set with the lowest ping time.
- construct a list of replica set members that is within a ping time of 15 milliseconds of the nearest suitable member of the set.

  If you specify a value for `--localThreshold` (page 591), `mongos` (page 676) will construct the list of replica members that are within the latency allowed by this value.
- The `mongos` (page 676) will select a member to read from at random from this list.

The ping time used for a set member compared by the `--localThreshold` (page 591) setting is a moving average of recent ping times, calculated, at most, every 10 seconds. As a result, some queries may reach members above the threshold until the `mongos` (page 676) recalculates the average.

See the *Member Selection* (page 60) section of the *read preference* (page 55) documentation for more information.

mongo

Synopsis

```
mongo [-shell] [-nodb] [-norc] [-quiet] [-port <port>] [-host <host>] [-eval <JavaScript>]
```

Description

mongo

`mongo` is an interactive JavaScript shell interface to MongoDB. The `mongo` command provides a powerful interface for systems administrators as well as a way to test queries and operations directly with the database. To increase the flexibility of the `mongo` command, the shell provides a fully functional JavaScript environment. This document addresses the basic invocation of the `mongo` shell and an overview of its usage.

Interface

Options

--shell
Enables the shell interface after evaluating a *JavaScript* file. If you invoke the `mongo` command and specify a JavaScript file as an argument, or use `mongo --eval` (page 592) to specify JavaScript on the command line, the `mongo --shell` (page 591) option provides the user with a shell prompt after the file finishes executing.

--nodb
Prevents the shell from connecting to any database instances.

--norc
Prevents the shell from sourcing and evaluating `~/.mongorc.js` on startup.

--quiet
Silences output from the shell during the connection process.

--port <PORT>
Specifies the port where the `mongod` or `mongos` (page 676) instance is listening. Unless specified `mongo` connects to `mongod` instances on port 27017, which is the default `mongod` port.

--host <HOSTNAME>
specifies the host where the `mongod` or `mongos` (page 676) is running to connect to as <HOSTNAME>. By default `mongo` will attempt to connect to a MongoDB process running on the localhost.

--eval <JAVASCRIPT>
Evaluates a JavaScript expression specified as an argument to this option. `mongo` does not load its own environment when evaluating code: as a result many options of the shell environment are not available.

--username <USERNAME>, **-u** <USERNAME>
Specifies a username to authenticate to the MongoDB instance. Use in conjunction with the *mongo --password* (page 592) option to supply a password. If you specify a username and password but the default database or the specified database do not require authentication, `mongo` will exit with an exception.

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongo --username* (page 592) option to supply a username. If you specify a *--username* (page 592) without the *mongo --password* (page 592) option, `mongo` will prompt for a password interactively, if the `mongod` or `mongos` (page 676) requires authentication.

--help, -h
Returns a basic help and usage text.

--version
Returns the version of the shell.

--verbose
Increases the verbosity of the output of the shell during the connection process.

--ipv6
Enables IPv6 support that allows `mongo` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongo`, disable IPv6 support by default.

<db address>
Specifies the "database address" of the database to connect to. For example:

```
mongo admin
```

The above command will connect the `mongo` shell to the *admin database* on the local machine. You may specify a remote database instance, with the resolvable hostname or IP address. Separate the database name from the hostname using a `http://docs.mongodb.org/manual/` character. See the following examples:

```
mongo mongodb1.example.net
mongo mongodb1/admin
mongo 10.8.8.10/test
```

<file.js>
Specifies a JavaScript file to run and then exit. Must be the last option specified. Use the *mongo --shell* (page 591) option to return to a shell after the file finishes running.

Files `~/.dbshell`

`mongo` maintains a history of commands in the `.dbshell` file.

Note: Interaction related to authentication, including `authenticate` and `db.addUser()` are not saved in the history file.

Warning: Versions of Windows **mongo.exe** earlier than 2.2.0 will save the *.dbshell* file in the **mongo.exe** working directory.

`~/.mongorc.js`

`mongo` will read *.mongorc.js* from the home directory of the user invoking `mongo`. Specify the *mongo --norc* (page 591) option to disable reading *.mongorc.js*.

`http://docs.mongodb.org/manual/tmp/mongo_edit<time_t>.js`

Created by `mongo` when editing a file. If the file exists `mongo` will append an integer from `1` to `10` to the time value to attempt to create a unique file.

`%TEMP%mongo_edit<time_t>.js`

Created by **mongo.exe** on Windows when editing a file. If the file exists `mongo` will append an integer from `1` to `10` to the time value to attempt to create a unique file.

Environment

EDITOR

Specifies the path to an editor to use with the *edit* shell command. A JavaScript variable *EDITOR* will override the value of `EDITOR` (page 593).

HOME

Specifies the path to the home directory where `mongo` `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

HOMEDRIVE

On Windows systems, `HOMEDRIVE` (page 593) specifies the path the directory where `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

HOMEPATH

Specifies the Windows path to the home directory where `mongo` will read the `.mongorc.js` file and write the `.dbshell` file.

Use

Typically users invoke the shell with the `mongo` command at the system prompt. Consider the following examples for other scenarios.

To connect to a database on a remote host using authentication and a non-standard port, use the following form:

```
mongo --username <user> --password <pass> --hostname <host> --port 28015
```

Alternatively, consider the following short form:

```
mongo -u <user> -p <pass> --host <host> --port 28015
```

Replace `<user>`, `<pass>`, and `<host>` with the appropriate values for your situation and substitute or omit the *--port* (page 592) as needed.

To execute a JavaScript file without evaluating the `~/.mongorc.js` file before starting a shell session, use the following form:

```
mongo --shell --norc alternate-environment.js
```

To print return a query as *JSON*, from the system prompt using the *--eval* (page 592) option, use the following form:

```
mongo --eval 'db.collection.find().forEach(printjson)'
```

Use single quotes (e.g. `'`) to enclose the JavaScript, as well as the additional JavaScript required to generate this output.

37.1.2 Windows Services

The `mongod.exe` and `mongos.exe` describe the options available for configuring MongoDB when running as a Windows Service. The `mongod.exe` and `mongos.exe` binaries provide a superset of the `mongod` and `mongos` (page 676) options.

mongod.exe

Synopsis

`mongod.exe` is the build of the MongoDB daemon (i.e. `mongod`) for the Windows platform. `mongod.exe` has all of the features of `mongod` on Unix-like platforms and is completely compatible with the other builds of `mongod`. In addition, `mongod.exe` provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongod.exe`. All `mongod` options are available. See the "*mongod* (page 581)" and the "*Configuration File Options* (page 621)" documents for more information regarding `mongod.exe`.

To install and use `mongod.exe`, read the "*Install MongoDB on Windows* (page 22)" document.

Options

--install
Installs `mongod.exe` as a Windows Service and exits.

--remove
Removes the `mongod.exe` Windows Service. If `mongod.exe` is running, this operation will stop and then remove the service.

Note: *--remove* (page 594) requires the *--serviceName* (page 594) if you configured a non-default *--serviceName* (page 594) during the *--install* (page 594) operation.

--reinstall
Removes `mongod.exe` and reinstalls `mongod.exe` as a Windows Service.

--serviceName <name>
Default: "MongoDB"

Set the service name of `mongod.exe` when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *--serviceName* (page 594) in conjunction with either the *--install* (page 594) or *--remove* (page 594) install option.

--serviceDisplayName <name>
Default: "Mongo DB"

Sets the name listed for MongoDB on the Services administrative application.

--serviceDescription <description>
Default: "MongoDB Server"

Sets the `mongod.exe` service description.

You must use *--serviceDescription* (page 595) in conjunction with the *--install* (page 594) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

--serviceUser <user>
Runs the `mongod.exe` service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 595) in conjunction with the *--install* (page 594) option.

--servicePassword <password>
Sets the password for <user> for `mongod.exe` when running with the *--serviceUser* (page 595) option.

You must use *--servicePassword* (page 595) in conjunction with the *--install* (page 594) option.

mongos.exe

Synopsis

`mongos.exe` is the build of the MongoDB Shard (i.e. `mongos` (page 676)) for the Windows platform. `mongos.exe` has all of the features of `mongos` (page 676) on Unix-like platforms and is completely compatible with the other builds of `mongos` (page 676). In addition, `mongos.exe` provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongos.exe`. All `mongos` (page 676) options are available. See the "*mongos* (page 588)" and the "*Configuration File Options* (page 621)" documents for more information regarding `mongos.exe`.

To install and use `mongos.exe`, read the "*Install MongoDB on Windows* (page 22)" document.

Options

--install
Installs `mongos.exe` as a Windows Service and exits.

--remove
Removes the `mongos.exe` Windows Service. If `mongos.exe` is running, this operation will stop and then remove the service.

Note: *--remove* (page 595) requires the *--serviceName* (page 596) if you configured a non-default *--serviceName* (page 596) during the *--install* (page 595) operation.

--reinstall
Removes `mongos.exe` and reinstalls `mongos.exe` as a Windows Service.

--serviceName <name>
Default: "MongoS"

Set the service name of `mongos.exe` when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *--serviceName* (page 596) in conjunction with either the *--install* (page 595) or *--remove* (page 595) install option.

--serviceDisplayName <name>
Default: "Mongo DB Router"

Sets the name listed for MongoDB on the Services administrative application.

--serviceDescription <description>
Default: "Mongo DB Sharding Router"

Sets the `mongos.exe` service description.

You must use *--serviceDescription* (page 596) in conjunction with the *--install* (page 595) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

--serviceUser <user>
Runs the `mongos.exe` service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 596) in conjunction with the *--install* (page 595) option.

--servicePassword <password>
Sets the password for <user> for `mongos.exe` when running with the *--serviceUser* (page 596) option.

You must use *--servicePassword* (page 596) in conjunction with the *--install* (page 595) option.

37.1.3 Binary Import and Export Tools

`mongodump` provides a method for creating *BSON* dump files from the `mongod` instances, while `mongorestore` makes it possible to restore these dumps. `bsondump` converts BSON dump files into *JSON*. The `mongooplog` utility provides the ability to stream *oplog* entries outside of normal replication.

mongodump

Synopsis

`mongodump` is a utility for creating a binary export of the contents of a database. Consider using this utility as part an effective *backup strategy* (page 180). Use in conjunction with `mongorestore` to provide restore functionality.

Note: If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you can restore that dump only to a 2.2 database.

See Also:

"`mongorestore`" and "*Backup and Restoration Strategies* (page 180)".

Options

mongodump

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongodump` utility and exits.

--host <hostname><:port>
Specifies a resolvable hostname for the `mongod` that you wish to use to create the database dump. By default `mongodump` will attempt to connect to a MongoDB process ruining on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, use the *--host* (page 597) argument with a setname, followed by a slash and a comma separated list of host names and port numbers. The `mongodump` utility will, given the seed of at least one connected set member, connect to the primary member of that set. This option would resemble:

```
mongodump --host repl0/mongo0.example.net,mongo0.example.net:27018,mongo1.example.net,mongo2.exa
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *--host* (page 597) option.

--ipv6
Enables IPv6 support that allows `mongodump` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongodump`, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 597) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 597) option to supply a username.

If you specify a *--username* (page 597) without the *--password* (page 597) option, `mongodump` will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 597) option enables `mongodump` to attach directly to local data files and copy the data without the `mongod`. To run with *--dbpath* (page 597), `mongodump` needs to restrict access to the data directory: as a result, no `mongod` can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 597) in conjunction with the corresponding option to `mongod`. This option allows `mongodump` to read data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 597) option.

--journal
Allows `mongodump` operations to use the durability *journal* to ensure that the export is in a consistent state. This option is only relevant when specifying the *--dbpath* (page 597) option.

--db <db>, **-d** <db>
Use the *--db* (page 597) option to specify a database for mongodump to backup. If you do not specify a DB, mongodump copies all databases in this instance into the dump files. Use this option to backup or copy a smaller subset of your data.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 598) option to specify a collection for mongodump to backup. If you do not specify a collection, this option copies all collections in the specified database or instance to the dump files. Use this option to backup or copy a smaller subset of your data.

--out <path>, **-o** <path>
Specifies a path where mongodump and store the output the database dump. To output the database dump to standard output, specify a – rather than a path.

--query <json>, **-q** <json>
Provides a query to limit (optionally) the documents included in the output of mongodump.

--oplog
Use this option to ensure that mongodump creates a dump of the database that includes an *oplog*, to create a point-in-time snapshot of the state of a mongod instance. To restore to a specific point-in-time backup, use the output created with this option in conjunction with *mongorestore --oplogReplay* (page 601).

Without *--oplog* (page 598), if there are write operations during the dump operation, the dump will not reflect a single moment in time. Changes made to the database during the update process can affect the output of the backup.

--oplog (page 598) has no effect when running mongodump against a mongos (page 676) instance to dump the entire contents of a sharded cluster. However, you can use *--oplog* (page 598) to dump individual shards.

Note: *--oplog* (page 598) only works against nodes that maintain a *oplog*. This includes all members of a replica set, as well as *master* nodes in master/slave replication deployments.

--repair
Use this option to run a repair option in addition to dumping the database. The repair option attempts to repair a database that may be in an inconsistent state as a result of an improper shutdown or mongod crash.

--forceTableScan
Forces mongodump to scan the data store directly: typically, mongodump saves entries as they appear in the index of the _id field. Use *--forceTableScan* (page 598) to skip the index and scan the data directly. Typically there are two cases where this behavior is preferable to the default:

1. If you have key sizes over 800 bytes that would not be present in the _id index.
2. Your database uses a custom _id field.

When you run with *--forceTableScan* (page 598), mongodump does not use $snapshot (page 464). As a result, the dump produced by mongodump can reflect the state of the database at many different points in time.

Warning: Use *--forceTableScan* (page 598) with extreme caution and consideration.

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.

Behavior

When running mongodump against a mongos (page 676) instance where the *sharded cluster* consists of *replica sets*, the *read preference* of the operation will prefer reads from *secondary* members of the set.

Usage

See the "*backup guide section on database dumps* (page 185)" for a larger overview of mongodump usage. Also see the "*mongorestore* (page 599)" document for an overview of the mongorestore, which provides the related inverse functionality.

The following command, creates a dump file that contains only the collection named collection in the database named test. In this case the database is running on the local interface on port 27017:

```
mongodump --collection collection --db test
```

In the next example, mongodump creates a backup of the database instance stored in the http://docs.mongodb.org/manual/srv/mongodb directory on the local machine. This requires that no mongod instance is using the http://docs.mongodb.org/manual/srv/mongodb directory.

```
mongodump --dbpath /srv/mongodb
```

In the final example, mongodump creates a database dump located at http://docs.mongodb.org/manual/opt/backup/mongodump-2011-10-24, from a database running on port 37017 on the host mongodb1.example.net and authenticating using the username user and the password pass, as follows:

```
mongodump --host mongodb1.example.net --port 37017 --username user --password pass /opt/backup/mongod
```

mongorestore

Synopsis

The mongorestore tool imports content from binary database dump, created by mongodump into a specific database. mongorestore can import content to an existing database or create a new one.

mongorestore, and only performs inserts into the existing database, and does not perform updates or *upserts*. If existing data with the same _id already exists on the target database, mongorestore will *not* replace it.

mongorestore will recreate indexes from the dump

The behavior of mongorestore has the following properties:

- all operations are inserts, not updates.
- all inserts are "fire and forget," mongorestore does not wait for a response from a mongod to ensure that the MongoDB process has received or recorded the operation.

  The mongod will record any errors to its log that occur during a restore operation but mongorestore will not receive errors.

Note: If you use the mongodump tool from the 2.2 distribution to create a dump of a database, you can restore that dump only to a 2.2 database.

Options

mongorestore

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongorestore` tool.

--host <hostname><:port>
Specifies a resolvable hostname for the `mongod` to which you want to restore the database. By default `mongorestore` will attempt to connect to a MongoDB process running on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *--host* (page 600) command.

--ipv6
Enables IPv6 support that allows `mongorestore` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongorestore`, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 600) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongorestore --username* (page 600) option to supply a username.

If you specify a *--username* (page 600) without the *--password* (page 600) option, `mongorestore` will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 600) option enables `mongorestore` to attach directly to local data files and insert the data without the `mongod`. To run with *--dbpath* (page 600), `mongorestore` needs to lock access to the data directory: as a result, no `mongod` can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 600) in conjunction with the corresponding option to `mongod`, which allows `mongorestore` to import data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 600) option.

--journal
Allows `mongorestore` write to the durability *journal* to ensure that the data files will remain in a consistent state during the write process. This option is only relevant when specifying the *--dbpath* (page 600) option.

--db <db>, **-d** <db>
Use the *--db* (page 600) option to specify a database for `mongorestore` to restore data *into*. If the database doesn't exist, `mongorestore` will create the specified database. If you do not specify a <db>, `mongorestore` creates new databases that correspond to the databases where data originated and data may be overwritten. Use this option to restore data into a MongoDB instance that already has data.

--db (page 600) does *not* control which *BSON* files `mongorestore` restores. You must use the `mongorestore` *path option* (page 601) to limit that restored data.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 601) option to specify a collection for `mongorestore` to restore. If you do not specify a <collection>, `mongorestore` imports all collections created. Existing data may be overwritten. Use this option to restore data into a MongoDB instance that already has data, or to restore only some data in the specified imported data set.

--objcheck
Verifies each object as a valid *BSON* object before inserting it into the target database. If the object is not a valid *BSON* object, `mongorestore` will not insert the object into the target database and stop processing remaining documents for import. This option has some performance impact.

--filter '<JSON>'
Limits the documents that `mongorestore` imports to only those documents that match the JSON document specified as '<JSON>'. Be sure to include the document in single quotes to avoid interaction with your system's shell environment.

--drop
Modifies the restoration procedure to drop every collection from the target database before restoring the collection from the dumped backup.

--oplogReplay
Replays the *oplog* after restoring the dump to ensure that the current state of the database reflects the point-in-time backup captured with the "*mongodump --oplog* (page 598)" command.

--keepIndexVersion
Prevents `mongorestore` from upgrading the index to the latest version during the restoration process.

--w <number of replicas per write>
New in version 2.2. Specifies the *write concern* (page 54) for each write operation that `mongorestore` writes to the target database. By default, `mongorestore` waits for the write operation to return on 1 member of the set (i.e. the *primary*.)

--noOptionsRestore
New in version 2.2. Prevents `mongorestore` from setting the collection options, such as those specified by the `collMod` *database command*, on restored collections.

--noIndexRestore
New in version 2.2. Prevents `mongorestore` from restoring and building indexes as specified in the corresponding `mongodump` output.

--oplogLimit <timestamp>
New in version 2.2. Prevents `mongorestore` from applying *oplog* entries newer than the <timestamp>. Specify <timestamp> values in the form of <time_t>:<ordinal>, where <time_t> is the seconds since the UNIX epoch, and <ordinal> represents a counter of operations in the oplog that occurred in the specified second.

You must use *--oplogLimit* (page 601) in conjunction with the *--oplogReplay* (page 601) option.

<path>
The final argument of the `mongorestore` command is a directory path. This argument specifies the location of the database dump from which to restore.

Usage

See the "*backup guide section on database dumps* (page 185)" for a larger overview of `mongorestore` usage. Also see the "*mongodump* (page 596)" document for an overview of the `mongodump`, which provides the related inverse

functionality.

Consider the following example:

```
mongorestore --collection people --db accounts dump/accounts/
```

Here, mongorestore reads the database dump in the dump/ sub-directory of the current directory, and restores *only* the documents in the collection named people from the database named accounts. mongorestore restores data to the instance running on the localhost interface on port 27017.

In the next example, mongorestore restores a backup of the database instance located in dump to a database instance stored in the http://docs.mongodb.org/manual/srv/mongodb on the local machine. This requires that there are no active mongod instances attached to http://docs.mongodb.org/manual/srv/mongodb data directory.

```
mongorestore --dbpath /srv/mongodb
```

In the final example, mongorestore restores a database dump located at http://docs.mongodb.org/manual/opt/backup/mongodump-2011-10-24, from a database running on port 37017 on the host mongodb1.example.net. mongorestore' authenticates to the this MongoDB instance using the username user and the password pass, as follows:

```
mongorestore --host mongodb1.example.net --port 37017 --username user --password pass /opt/backup/mor
```

bsondump

Synopsis

The bsondump converts *BSON* files into human-readable formats, including *JSON*. For example, bsondump is useful for reading the output files generated by mongodump.

Options

bsondump

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the bsondump utility.

--objcheck
Validates each *BSON* object before outputting it in *JSON* format. Use this option to filter corrupt objects from the output. This option has some performance impact.

--filter '<JSON>'
Limits the documents that bsondump exports to only those documents that match the *JSON document* specified as '<JSON>'. Be sure to include the document in single quotes to avoid interaction with your system's shell environment.

--type <=json|=debug>
Changes the operation of bsondump from outputting "*JSON*" (the default) to a debugging format.

<bsonfilename>
The final argument to bsondump is a document containing *BSON*. This data is typically generated by mongodump or by MongoDB in a *rollback* operation.

Usage

By default, bsondump outputs data to standard output. To create corresponding *JSON* files, you will need to use the shell redirect. See the following command:

```
bsondump collection.bson > collection.json
```

Use the following command (at the system shell) to produce debugging output for a *BSON* file:

```
bsondump --type=debug collection.bson
```

mongooplog

New in version 2.1.1.

Synopsis

mongooplog is a simple tool that polls operations from the *replication oplog* of a remote server, and applies them to the local server. This capability supports certain classes of real-time migrations that require that the source server remain online and in operation throughout the migration process.

Typically this command will take the following form:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

This command copies oplog entries from the mongod instance running on the host mongodb0.example.net and duplicates operations to the host mongodb1.example.net. If you do not need to keep the *--from* (page 605) host running during the migration, consider using mongodump and mongorestore or another *backup* (page 180) operation, which may be better suited to your operation.

Note: If the mongod instance specified by the *--from* (page 605) argument is running with authentication (page 624), then mongooplog will not be able to copy oplog entries.

See Also:

mongodump, mongorestore, "*Backup and Restoration Strategies* (page 180)," "*Oplog Internals Overview* (page 62), and "*Replica Set Oplog Sizing* (page 36)".

Options

mongooplog

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the -v form by including the option multiple times, (e.g. -vvvvv.)

--version
Returns the version of the mongooplog utility.

--host <hostname><:port>, **-h**
Specifies a resolvable hostname for the mongod instance to which mongooplog will apply *oplog* operations retrieved from the serve specified by the *--from* (page 605) option.

mongooplog assumes that all target mongod instances are accessible by way of port 27017. You may, optionally, declare an alternate port number as part of the hostname argument.

You can always connect directly to a single mongod instance by specifying the host and port number directly.

--port
Specifies the port number of the mongod instance where mongooplog will apply *oplog* entries. Only specify this option if the MongoDB instance that you wish to connect to is not running on the standard port. (i.e. 27017) You may also specify a port number using the *--host* (page 604) command.

--ipv6
Enables IPv6 support that allows mongooplog to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongooplog, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* (page 604) option to supply a password.

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 604) option to supply a username.

If you specify a *--username* (page 604) without the *--password* (page 604) option, mongooplog will prompt for a password interactively.

--dbpath <path>
Specifies a directory, containing MongoDB data files, to which mongooplog will apply operations from the *oplog* of the database specified with the *--from* (page 605) option. When used, the *--dbpath* (page 604) option enables mongo to attach directly to local data files and write data without a running mongod instance. To run with *--dbpath* (page 604), mongooplog needs to restrict access to the data directory: as a result, no mongod can be access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 604) in conjunction with the corresponding option to mongod. This option allows mongooplog to write to data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 604) option.

--journal
Allows mongooplog operations to use the durability *journal* to ensure that the data files will remain in a consistent state during the writing process. This option is only relevant when specifying the *--dbpath* (page 604) option.

--db <db>, **-d** <db>
Use the *--db* (page 604) option to specify a database for mongooplog to write data to. If you do not specify a DB, mongooplog will apply operations that apply to all databases that appear in the oplog. Use this option to migrate a smaller subset of your data.

--collection <collection>, **-c** <c>
Use the *--collection* (page 604) option to specify a collection for mongooplog to write data to. If you do not specify a collection, mongooplog will apply operations that apply to all collections that appear in the oplog in the specified database. Use this option to migrate a smaller subset of your data.

--fields [field1[,field2]], **-f** [field1[,field2]]
Specify a field or number fields to constrain which data mongooplog will migrate. All other fields will be *excluded* from the migration. Comma separate a list of fields to limit the applied fields.

--fieldFile <file>
As an alternative to "*--fields* (page 604)" the *--fieldFile* (page 605) option allows you to specify a file (e.g. <file>) that holds a list of field names to *include* in the migration. All other fields will be *excluded* from the migration. Place one field per line.

--seconds <number>, **-s** <number>
Specify a number of seconds of operations for mongooplog to pull from the *remote host* (page 605). Unless specified the default value is 86400 seconds, or 24 hours.

--from <host[:port]>
Specify the host for mongooplog to retrieve *oplog* operations from. mongooplog *requires* this option.

Unless you specify the *--host* (page 604) option, mongooplog will apply the operations collected with this option to the oplog of the mongod instance running on the localhost interface connected to port 27017.

--oplogns <namespace>
Specify a namespace in the *--from* (page 605) host where the oplog resides. The default value is local.oplog.rs, which is the where *replica set* members store their operation log. However, if you've copied *oplog* entries into another database or collection, use this option to copy oplog entries stored in another location.

Namespaces take the form of [database].[collection].

Usage Consider the following prototype mongooplog command:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

Here, entries from the *oplog* of the mongod running on port 27017. This only pull entries from the last 24 hours.

In the next command, the parameters limit this operation to only apply operations to the database people in the collection usage on the target host (i.e. mongodb1.example.net):

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net --database people --collection us
```

This operation only applies oplog entries from the last 24 hours. Use the *--seconds* (page 605) argument to capture a greater or smaller amount of time. Consider the following example:

```
mongooplog  --from mongodb0.example.net --seconds 172800
```

In this operation, mongooplog captures 2 full days of operations. To migrate 12 hours of *oplog* entries, use the following form:

```
mongooplog  --from mongodb0.example.net --seconds 43200
```

For the previous two examples, mongooplog migrates entries to the mongod process running on the localhost interface connected to the 27017 port. mongooplog can also operate directly on MongoDB's data files if no mongod is running on the *target* host. Consider the following example:

```
mongooplog  --from mongodb0.example.net --dbpath /srv/mongodb --journal
```

Here, mongooplog imports *oplog* operations from the mongod host connected to port 27017. This migrates operations to the MongoDB data files stored in the http://docs.mongodb.org/manual/srv/mongodb directory. Additionally mongooplog will use the durability *journal* to ensure that the data files remain in a consistent state.

37.1.4 Data Import and Export Tools

`mongoimport` provides a method for taking data in *JSON*, *CSV*, or *TSV* and importing it into a `mongod` instance. `mongoexport` provides a method to export data from a `mongod` instance into JSON, CSV, or TSV.

Note: The conversion between BSON and other formats lacks full type fidelity. Therefore you cannot use `mongoimport` and `mongoexport` for round-trip import and export operations.

---

mongoimport

Synopsis

The `mongoimport` utility provides a route to import content from a JSON, CSV, or TSV export created by `mongoexport`, or potentially, another third-party export tool. See the "*Importing and Exporting MongoDB Data* (page 177)" document for a more in depth usage overview, and the "*mongoexport* (page 609)" document for more information regarding the `mongoexport` utility, which provides the inverse "importing" capability.

Note: Do not use `mongoimport` and `mongoexport` for full instance, production backups because they will not reliably capture data type information. Use `mongodump` and `mongorestore` as described in "*Backup and Restoration Strategies* (page 180)" for this kind of functionality.

Options

mongoimport

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongoimport` utility.

--host `<hostname><:port>`, **-h**
Specifies a resolvable hostname for the `mongod` to which you want to restore the database. By default `mongoimport` will attempt to connect to a MongoDB process ruining on the localhost port numbered `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, use the *--host* (page 606) argument with a setname, followed by a slash and a comma separated list of host and port names. The `mongo` utility will, given the seed of at least one connected set member, connect to primary node of that set. this option would resemble:

```
--host repl0/mongo0.example.net,mongo0.example.net,27018,mongo1.example.net,mongo2.example.net
```

You can always connect directly to a single MongoDB instance by specifying the host and port number directly.

--port `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *mongoimport --host* (page 606) command.

--ipv6
Enables IPv6 support that allows mongoimport to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongoimport, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongoimport --password* (page 607) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongoimport --username* (page 607) option to supply a username.

If you specify a *--username* (page 607) without the *--password* (page 607) option, mongoimport will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 607) option enables mongoimport to attach directly to local data files and insert the data without the mongod. To run with --dbpath, mongoimport needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 607) in conjunction with the corresponding option to mongod, which allows mongoimport to import data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 607) option.

--journal
Allows mongoexport write to the durability *journal* to ensure that the data files will remain in a consistent state during the write process. This option is only relevant when specifying the *--dbpath* (page 607) option.

--db <db>, **-d** <db>
Use the *--db* (page 607) option to specify a database for mongoimport to restore data. If you do not specify a <db>, mongoimport creates new databases that correspond to the databases where data originated and data may be overwritten. Use this option to restore data into a MongoDB instance that already has data, or to restore only some data in the specified backup.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 607) option to specify a collection for mongorestore to restore. If you do not specify a <collection>, mongoimport imports all collections created. Existing data may be overwritten. Use this option to restore data into a MongoDB instance that already has data, or to restore only some data in the specified imported data set.

--fields <field1[,filed2]>, **-f** <field1[,filed2]>
Specify a field or number fields to *import* from the specified file. All other fields present in the export will be *excluded* during importation. Comma separate a list of fields to limit the fields imported.

--fieldFile <filename>
As an alternative to "*mongoimport --fields* (page 607)" the *--fieldFile* (page 607) option allows you to specify a file (e.g. <file> ') to hold a list of field names to specify a list of fields to *include* in the export. All other fields will be *excluded* from the export. Place one field per line.

--ignoreBlanks
In *csv* and *tsv* exports, ignore empty fields. If not specified, mongoimport creates fields without values in imported documents.

--type <json|csv|tsv>
Declare the type of export format to import. The default format is *JSON*, but it's possible to import *csv* and *tsv* files.

--file <filename>
Specify the location of a file containing the data to import. mongoimport will read data from standard input (e.g. "stdin.") if you do not specify a file.

--drop
Modifies the importation procedure so that the target instance drops every collection before restoring the collection from the dumped backup.

--headerline
If using "*--type csv* (page 607)" or "*--type tsv* (page 607)," use the first line as field names. Otherwise, mongoimport will import the first line as a distinct document.

--upsert
Modifies the import process to update existing objects in the database if they match an imported object, while inserting all other objects.

If you do not specify a field or fields using the *--upsertFields* (page 608) mongoimport will upsert on the basis of the _id field.

--upsertFields <field1[,field2]>
Specifies a list of fields for the query portion of the *upsert*. Use this option if the _id fields in the existing documents don't match the field in the document, but another field or field combination can uniquely identify documents as a basis for performing upsert operations.

To ensure adequate performance, indexes should exist for this field or fields.

--stopOnError
New in version 2.2. Forces mongoimport to halt the import operation at the first error rather than continuing the operation despite errors.

--jsonArray
Changed in version 2.2: The limit on document size increased from 4MB to 16MB. Accept import of data expressed with multiple MongoDB document within a single *JSON* array.

Use in conjunction with *mongoexport --jsonArray* (page 610) to import data written as a single *JSON* array. Limited to imports of 16 MB or smaller.

Usage

In this example, mongoimport imports the *csv* formatted data in the http://docs.mongodb.org/manual/opt/backups/contacts.csv into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017.

```
mongoimport --db users --collection contacts --type csv --file /opt/backups/contacts.csv
```

In the following example, mongoimport imports the data in the *JSON* formatted file contacts.json into the collection contacts on the MongoDB instance running on the localhost port number 27017. Journaling is explicitly enabled.

```
mongoimport --collection contacts --file contacts.json --journal
```

In the next example, mongoimport takes data passed to it on standard input (i.e. with a | pipe.) and imports it into the collection contacts in the sales database is the MongoDB datafiles located at http://docs.mongodb.org/manual/srv/mongodb/. if the import process encounters an error, the mongoimport will halt because of the *--stopOnError* (page 608) option.

```
mongoimport --db sales --collection contacts --stopOnError --dbpath /srv/mongodb/
```

In the final example, `mongoimport` imports data from the file `http://docs.mongodb.org/manual/opt/backups/mdb1-e` into the collection `contacts` within the database `marketing` on a remote MongoDB database. This `mongoimport` accesses the `mongod` instance running on the host `mongodb1.example.net` over port `37017`, which requires the username `user` and the password `pass`.

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

mongoexport

Synopsis

`mongoexport` is a utility that produces a JSON or CSV export of data stored in a MongoDB instance. See the "*Importing and Exporting MongoDB Data* (page 177)" document for a more in depth usage overview, and the "*mongoimport* (page 606)" document for more information regarding the `mongoimport` utility, which provides the inverse "importing" capability.

Note: Do not use `mongoimport` and `mongoexport` for full-scale backups because they may not reliably capture data type information. Use `mongodump` and `mongorestore` as described in "*Backup and Restoration Strategies* (page 180)" for this kind of functionality.

Options

mongoexport

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongoexport` utility.

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` from which you want to export data. By default `mongoexport` attempts to connect to a MongoDB process ruining on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

--port `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *mongoexport --host* (page 609) command.

--ipv6
Enables IPv6 support that allows `mongoexport` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongoexport`, disable IPv6 support by default.

--username `<username>`**, -u** `<username>`
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongoexport --password* (page 609) option to supply a password.

--password `<password>`
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 609) option to supply a username.

If you specify a *--username* (page 609) without the *--password* (page 609) option, mongoexport will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the --dbpath option enables mongoexport to attach directly to local data files and insert the data without the mongod. To run with --dbpath, mongoexport needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 610) in conjunction with the corresponding option to mongod, which allows mongoexport to export data into MongoDB instances that have every database's files saved in discrete directories on the disk. This option is only relevant when specifying the *--dbpath* (page 610) option.

--journal
Allows mongoexport operations to access the durability *journal* to ensure that the export is in a consistent state. This option is only relevant when specifying the *--dbpath* (page 610) option.

--db <db>, **-d** <db>
Use the *--db* (page 610) option to specify the name of the database that contains the collection you want to export.

--collection <collection>, **-c** <collection>
Use the *--collection* (page 610) option to specify the collection that you want mongoexport to export.

--fields <field1[,field2]>, **-f** <field1[,field2]>
Specify a field or number fields to *include* in the export. All other fields will be *excluded* from the export. Comma separate a list of fields to limit the fields exported.

--fieldFile <file>
As an alternative to "*--fields* (page 610)" the *--fieldFile* (page 610) option allows you to specify a file (e.g. <file> ') to hold a list of field names to specify a list of fields to *include* in the export. All other fields will be *excluded* from the export. Place one field per line.

--query <JSON>
Provides a *JSON document* as a query that optionally limits the documents returned in the export.

--csv
Changes the export format to a comma separated values (CSV) format. By default mongoexport writes data using one *JSON* document for every MongoDB document.

--jsonArray
Modifies the output of mongoexport to write the entire contents of the export as a single *JSON* array. By default mongoexport writes data using one JSON document for every MongoDB document.

--slaveOk, -k
Allows mongoexport to read data from secondary or slave nodes when using mongoexport with a replica set. This option is only available if connected to a mongod or mongos (page 676) and is not available when used with the "*mongoexport --dbpath* (page 610)" option.

This is the default behavior.

--out <file>, **-o** <file>
Specify a file to write the export to. If you do not specify a file name, the mongoexport writes data to standard output (e.g. stdout).

Usage

In the following example, mongoexport exports the collection contacts from the users database from the mongod instance running on the localhost port number 27017. This command writes the export data in *CSV* format

into a file located at `http://docs.mongodb.org/manual/opt/backups/contacts.csv`.

```
mongoexport --db users --collection contacts --csv --out /opt/backups/contacts.csv
```

The next example creates an export of the collection `contacts` from the MongoDB instance running on the localhost port number `27017`, with journaling explicitly enabled. This writes the export to the `contacts.json` file in *JSON* format.

```
mongoexport --db sales --collection contacts --out contacts.json --journal
```

The following example exports the collection `contacts` from the `sales` database located in the MongoDB data files located at `http://docs.mongodb.org/manual/srv/mongodb/`. This operation writes the export to standard output in *JSON* format.

```
mongoexport --db sales --collection contacts --dbpath /srv/mongodb/
```

Warning: The above example will only succeed if there is no `mongod` connected to the data files located in the `http://docs.mongodb.org/manual/srv/mongodb/` directory.

The final example exports the collection `contacts` from the database `marketing` . This data resides on the MongoDB instance located on the host `mongodb1.example.net` running on port `37017`, which requires the username `user` and the password `pass`.

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

37.1.5 Diagnostic Tools

`mongostat`, `mongotop`, and `mongosniff` provide diagnostic information related to the current operation of a `mongod` instance.

Note: Because `mongosniff` depends on *libpcap*, most distributions of MongoDB do *not* include `mongosniff`.

---

mongostat

Synopsis

The `mongostat` utility provides a quick overview of the status of a currently running `mongod` instance. `mongostat` is functionally similar to the UNIX/Linux file system utility `vmstat`, but provides data regarding `mongod` instances.

See Also:

For more information about monitoring MongoDB, see *Monitoring Database Systems* (page 170).

For more background on various other MongoDB status outputs see:

- *Server Status Reference* (page 637)
- *Replica Set Status Reference* (page 659)
- *Database Statistics Reference* (page 651)
- *Collection Statistics Reference* (page 653)

For an additional utility that provides MongoDB metrics see "*mongotop* (page 615)."

`mongostat` connects to the `mongod` process running on the local host interface on TCP port `27017`; however, `mongostat` can connect to any accessible remote MongoDB process.

Options

mongostat

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the `mongostat` utility.

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` from which you want to export data. By default `mongostat` attempts to connect to a MongoDB process running on the localhost port number `27017`.

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

--port `<port>`
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. `27017`) You may also specify a port number using the *mongostat --host* (page 612) command.

--ipv6
Enables IPv6 support that allows `mongostat` to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongostat`, disable IPv6 support by default.

--username `<username>`**, -u** `<username>`
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongostat --password* (page 612) option to supply a password.

--password `<password>`
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongostat --username* (page 612) option to supply a username.

If you specify a *--username* (page 612) without the *--password* (page 612) option, `mongostat` will prompt for a password interactively.

--noheaders
Disables the output of column or field names.

--rowcount `<number>`**, -n** `<number>`
Controls the number of rows to output. Use in conjunction with "**mongostat <sleeptime>**" to control the duration of a `mongostat` operation.

Unless specification, `mongostat` will return an infinite number of rows (e.g. value of `0`.)

--http
Configures `mongostat` to collect data using HTTP interface rather than a raw database connection.

--discover
With this option `mongostat` discovers and reports on statistics from all members of a *replica set* or *sharded cluster*. When connected to any member of a replica set, *--discover* (page 612) all non-*hidden members* of the replica set. When connected to a `mongos` (page 676), `mongostat` will return data from all *shards* in the

cluster. If a replica set provides a shard in the sharded cluster, `mongostat` will report on non-hidden members of that replica set.

The *`mongostat --host`* (page 612) option is not required but potentially useful in this case.

`--all`
Configures `mongostat` to return all optional *fields* (page 613).

`<sleeptime>`
The final argument the length of time, in seconds, that `mongostat` waits in between calls. By default `mongostat` returns one call every second.

`mongostat` returns values that reflect the operations over a 1 second period. For values of `<sleeptime>` greater than 1, `mongostat` averages data to reflect average operations per second.

Fields

`mongostat` returns values that reflect the operations over a 1 second period. When **mongostat <sleeptime>** has a value greater than 1, `mongostat` averages the statistics to reflect average operations per second.

`mongostat` outputs the following fields:

`inserts`
The number of objects inserted into the database per second. If followed by an asterisk (e.g. `*`), the datum refers to a replicated operation.

`query`
The number of query operations per second.

`update`
The number of update operations per second.

`delete`
The number of delete operations per second.

`getmore`
The number of get more (i.e. cursor batch) operations per second.

`command`
The number of commands per second. On *slave* and *secondary* systems, `mongostat` presents two values separated by a pipe character (e.g. `|`), in the form of `local|replicated` commands.

`flushes`
The number of *fsync* operations per second.

`mapped`
The total amount of data mapped in megabytes. This is the total data size at the time of the last `mongostat` call.

`size`
The amount of (virtual) memory used by the process at the time of the last `mongostat` call.

`res`
The amount of (resident) memory used by the process at the time of the last `mongostat` call.

`faults`
Changed in version 2.1. The number of page faults per second.

Before version 2.1 this value was only provided for MongoDB instances running on Linux hosts.

locked
The percent of time in a global write lock. Changed in version 2.2: The `locked db` field replaces the `locked %` field to more appropriate data regarding the database specific locks in version 2.2.

locked db
New in version 2.2. The percent of time in the per-database context-specific lock. `mongostat` will report the database that has spent the most time since the last `mongostat` call with a write lock.

This value represents the amount of time the database had a database specific lock *and* the time that the `mongod` spent in the global lock. Because of this, and the sampling method, you may see some values greater than 100%.

idx miss
The percent of index access attempts that required a page fault to load a btree node. This is a sampled value.

qr
The length of the queue of clients waiting to read data from the MongoDB instance.

qw
The length of the queue of clients waiting to write data from the MongoDB instance.

ar
The number of active clients performing read operations.

aw
The number of active clients performing write operations.

netIn
The amount of network traffic, in *bits*, received by the MongoDB.

This includes traffic from `mongostat` itself.

netOut
The amount of network traffic, in *bits*, sent by the MongoDB.

This includes traffic from `mongostat` itself.

conn
The total number of open connections.

set
The name, if applicable, of the replica set.

repl
The replication status of the node.

Value	Replication Type
M	*master*
SEC	*secondary*
REC	recovering
UNK	unknown
SLV	*slave*

Usage

In the first example, `mongostat` will return data every second for 20 seconds. `mongostat` collects data from the `mongod` instance running on the localhost interface on port 27017. All of the following invocations produce identical behavior:

```
mongostat --rowcount 20 1
mongostat --rowcount 20
```

```
mongostat -n 20 1
mongostat -n 20
```

In the next example, `mongostat` returns data every 5 minutes (or 300 seconds) for as long as the program runs. `mongostat` collects data from the `mongod` instance running on the localhost interface on port `27017`. Both of the following invocations produce identical behavior.

```
mongostat --rowcount 0 300
mongostat -n 0 300
mongostat 300
```

In the following example, `mongostat` returns data every 5 minutes for an hour (12 times.) `mongostat` collects data from the `mongod` instance running on the localhost interface on port 27017. Both of the following invocations produce identical behavior.

```
mongostat --rowcount 12 300
mongostat -n 12 300
```

In many cases, using the *`--discover`* (page 612) will help provide a more complete snapshot of the state of an entire group of machines. If a `mongos` (page 676) process connected to a *sharded cluster* is running on port `27017` of the local machine, you can use the following form to return statistics from all members of the cluster:

```
mongostat --discover
```

mongotop

Synopsis

`mongotop` provides a method to track the amount of time a MongoDB instance spends reading and writing data. `mongotop` provides statistics on a per-collection level. By default, `mongotop` returns values every second.

See Also:

For more information about monitoring MongoDB, see *Monitoring Database Systems* (page 170).

For additional background on various other MongoDB status outputs see:

- *Server Status Reference* (page 637)
- *Replica Set Status Reference* (page 659)
- *Database Statistics Reference* (page 651)
- *Collection Statistics Reference* (page 653)

For an additional utility that provides MongoDB metrics see "*mongostat* (page 611)."

Options

mongotop

`--help`
Returns a basic help and usage text.

`--verbose, -v`
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Print the version of the mongotop utility and exit.

--host <hostname><:port>
Specifies a resolvable hostname for the mongod from which you want to export data. By default mongotop attempts to connect to a MongoDB process running on the localhost port number 27017.

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. 27017) You may also specify a port number using the *mongotop --host* (page 616) command.

--ipv6
Enables IPv6 support that allows mongotop to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongotop, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongotop* (page 616) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 616) option to supply a username.

If you specify a *--username* (page 616) without the *--password* (page 616) option, mongotop will prompt for a password interactively.

--locks
New in version 2.2. Toggles the mode of mongotop to report on use of per-database *locks* (page 638). These data are useful for measuring concurrent operations and lock percentage.

<sleeptime>
The final argument is the length of time, in seconds, that mongotop waits in between calls. By default mongotop returns data every second.

Fields

mongotop returns time values specified in milliseconds (ms.)

mongotop only reports active namespaces or databases, depending on the *--locks* (page 616) option. If you don't see a database or collection, it has received no recent activity. You can issue a simple operation in the mongo shell to generate activity to affect the output of mongotop.

ns
Contains the database namespace, which combines the database name and collection. Changed in version 2.2: If you use the *--locks* (page 616), the ns (page 616) field does not appear in the mongotop output.

db
New in version 2.2. Contains the name of the database. The database named . refers to the global lock, rather than a specific database.

This field does not appear unless you have invoked mongotop with the *--locks* (page 616) option.

total
Provides the total amount of time that this mongod spent operating on this namespace.

read
Provides the amount of time that this mongod spent performing read operations on this namespace.

write
Provides the amount of time that this `mongod` spent performing write operations on this namespace.

<timestamp>
Provides a time stamp for the returned data.

Use

By default `mongotop` connects to the MongoDB instance running on the localhost port `27017`. However, `mongotop` can optionally connect to remote `mongod` instances. See the *mongotop options* (page 615) for more information.

To force `mongotop` to return less frequently specify a number, in seconds at the end of the command. In this example, `mongotop` will return every 15 seconds.

```
mongotop 15
```

This command produces the following output:

```
connected to: 127.0.0.1

                    ns       total        read       write            2012-08-13T15:45:40
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms

                    ns       total        read       write            2012-08-13T15:45:55
test.system.namespaces         0ms         0ms         0ms
  local.system.replset         0ms         0ms         0ms
  local.system.indexes         0ms         0ms         0ms
  admin.system.indexes         0ms         0ms         0ms
                admin.         0ms         0ms         0ms
```

To return return a `mongotop` report every 5 minutes, use the following command:

```
mongotop 300
```

To report the use of per-database locks, use *`mongotop --locks`* (page 616), which produces the following output:

```
$ mongotop --locks
connected to: 127.0.0.1

                  db       total        read       write          2012-08-13T16:33:34
               local         0ms         0ms         0ms
               admin         0ms         0ms         0ms
                   .         0ms         0ms         0ms
```

mongosniff

Synopsis

`mongosniff` provides a low-level operation tracing/sniffing view into database activity in real time. Think of `mongosniff` as a MongoDB-specific analogue of `tcpdump` for TCP/IP network traffic. Typically, `mongosniff` is most frequently used in driver development.

Note: `mongosniff` requires `libpcap` and is only available for Unix-like systems. Furthermore, the version distributed with the MongoDB binaries is dynamically linked against aversion 0.9 of `libpcap`. If your system has a different version of `libpcap`, you will need to compile `mongosniff` yourself or create a symbolic link pointing to `libpcap.so.0.9` to your local version of `libpcap`. Use an operation that resembles the following:

```
ln -s /usr/lib/libpcap.so.1.1.1 /usr/lib/libpcap.so.0.9
```

Change the path's and name of the shared library as needed.

As an alliterative to `mongosniff`, Wireshark, a popular network sniffing tool is capable of inspecting and parsing the MongoDB wire protocol.

Options

mongosniff

--help
Returns a basic help and usage text.

--forward <host>:<port>
Declares a host to forward all parsed requests that the `mongosniff` intercepts to another `mongod` instance and issue those operations on that database instance.

Specify the target host name and port in the `<host>:<port>` format.

--source <NET [interface]>, <FILE [filename]>, <DIAGLOG [filename]>
Specifies source material to inspect. Use `--source NET [interface]` to inspect traffic from a network interface (e.g. `eth0` or `lo`.) Use `--source FILE [filename]` to read captured packets in *pcap* format.

You may use the `--source DIAGLOG [filename]` option to read the output files produced by the *--diaglog* (page 583) option.

--objcheck
Modifies the behavior to *only* display invalid BSON objects and nothing else. Use this option for troubleshooting driver development. This option has some performance impact on the performance of `mongosniff`.

<port>
Specifies alternate ports to sniff for traffic. By default, `mongosniff` watches for MongoDB traffic on port 27017. Append multiple port numbers to the end of `mongosniff` to monitor traffic on multiple ports.

Usage

Use the following command to connect to a `mongod` or `mongos` (page 676) running on port 27017 *and* 27018 on the localhost interface:

```
mongosniff --source NET lo 27017 27018
```

Use the following command to only log invalid *BSON* objects for the `mongod` or `mongos` (page 676) running on the localhost interface and port 27018, for driver development and troubleshooting:

```
mongosniff --objcheck --source NET lo 27018
```

37.1.6 GridFS

`mongofiles` provides a command-line interact to a MongoDB *GridFS* storage system.

mongofiles

Synopsis

The `mongofiles` utility makes it possible to manipulate files stored in your MongoDB instance in *GridFS* objects from the command line. It is particularly useful as it provides an interface between objects stored in your file system and GridFS.

All `mongofiles` commands take arguments in three groups:

1. *Options* (page 619). You may use one or more of these options to control the behavior of `mongofiles`.
2. *Commands* (page 619). Use one of these commands to determine the action of `mongofiles`.
3. A file name representing either the name of a file on your system's file system, a GridFS object.

Like `mongodump`, `mongoexport`, `mongoimport`, and `mongorestore` `mongofiles` can access data stored in a MongoDB data directory without requiring a running `mongod` instance, if no other `mongod` is running.

Note: For *replica sets*, `mongofiles` can only read from the set's '*primary*.

Commands

mongofiles

list <prefix>
Lists the files in the GridFS store. The characters specified after `list` (e.g. `<prefix>`) optionally limit the list of returned items to files that begin with that string of characters.

search <string>
Lists the files in the GridFS store with names that match any portion of `<string>`.

put <filename>
Copy the specified file from the local file system into GridFS storage.

Here, `<filename>` refers to the name the object will have in GridFS, and `mongofiles` assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* (page 620) option.

get <filename>
Copy the specified file from GridFS storage to the local file system.

Here, `<filename>` refers to the name the object will have in GridFS, and `mongofiles` assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* (page 620) option.

delete <filename>
Delete the specified file from GridFS storage.

Options

--help
Returns a basic help and usage text.

--verbose, -v
Increases the amount of internal reporting returned on the command line. Increase the verbosity with the `-v` form by including the option multiple times, (e.g. `-vvvvv`.)

--version
Returns the version of the mongofiles utility.

--host <hostname><:port>
Specifies a resolvable hostname for the mongod that holds your GridFS system. By default mongofiles attempts to connect to a MongoDB process ruining on the localhost port number 27017.

Optionally, specify a port number to connect a MongoDB instance running on a port other than 27017.

--port <port>
Specifies the port number, if the MongoDB instance is not running on the standard port. (i.e. 27017) You may also specify a port number using the *mongofiles --host* (page 620) command.

--ipv6
Enables IPv6 support that allows mongofiles to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongofiles, disable IPv6 support by default.

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongofiles --password* (page 620) option to supply a password.

--password <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongofiles --username* (page 620) option to supply a username.

If you specify a *--username* (page 620) without the *--password* (page 620) option, mongofiles will prompt for a password interactively.

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 620) option enables mongofiles to attach directly to local data files interact with the GridFS data without the mongod. To run with *--dbpath* (page 620), mongofiles needs to lock access to the data directory: as a result, no mongod can access the same path while the process runs.

--directoryperdb
Use the *--directoryperdb* (page 620) in conjunction with the corresponding option to mongod, which allows mongofiles when running with the *--dbpath* (page 620) option and MongoDB uses an on-disk format where every database has a distinct directory. This option is only relevant when specifying the *--dbpath* (page 620) option.

--journal
Allows mongofiles operations to use the durability *journal* when running with *--dbpath* (page 620) to ensure that the database maintains a recoverable state. This forces mongofiles to record all data on disk regularly.

--db <db>, **-d** <db>
Use the *--db* (page 620) option to specify the MongoDB database that stores or will store the GridFS files.

--collection <collection>, **-c** <collection>
This option has no use in this context and a future release may remove it. See SERVER-4931 for more information.

--local <filename>, **-l** <filename>
Specifies the local filesystem name of a file for get and put operations.

In the **mongofiles put** and **mongofiles get** commands the required <filename> modifier refers to the name the object will have in GridFS. mongofiles assumes that this reflects the file's name on the local file system. This setting overrides this default.

--type <MIME>, t <MIME>
Provides the ability to specify a *MIME* type to describe the file inserted into GridFS storage. mongofiles

omits this option in the default operation.

Use only with **mongofiles put** operations.

--replace, -r
Alters the behavior of **mongofiles put** to replace existing GridFS objects with the specified local file, rather than adding an additional object with the same name.

In the default operation, files will not be overwritten by a **mongofiles put** option.

37.2 Configuration File Options

37.2.1 Synopsis

Administrators and users can control `mongod` or `mongos` (page 676) instances at runtime either directly from *mongod's command line arguments* (page 581) or using a configuration file.

While both methods are functionally equivalent and all settings are similar, the configuration file method is preferable. If you installed from a package and have started MongoDB using your system's *control script*, you're already using a configuration file.

To start `mongod` or `mongos` (page 676) using a config file, use one of the following forms:

```
mongod --config /etc/mongodb.conf
mongod -f /etc/mongodb.conf
mongos --config /srv/mongodb/mongos.conf
mongos -f /srv/mongodb/mongos.conf
```

Declare all settings in this file using the following form:

```
<setting> = <value>
```

New in version 2.0: *Before* version 2.0, Boolean (i.e. `true|false`) or "flag" parameters, register as true, if they appear in the configuration file, regardless of their value.

37.2.2 Settings

verbose
Default: false

Increases the amount of internal reporting returned on standard output or in the log file generated by `logpath` (page 622).

v
Default: false

Alternate form of `verbose` (page 621).

vv
Default: false

Additional increase in verbosity of output and logging.

vvv
Default: false

Additional increase in verbosity of output and logging.

vvvv

Default: false

Additional increase in verbosity of output and logging.

vvvvv

Default: false

Additional increase in verbosity of output and logging.

quiet

Default: false

Runs the `mongod` or `mongos` (page 676) instance in a quiet mode that attempts to limit the amount of output. This option suppresses:

- output from *database commands*, including `drop`, `dropIndex`, `diagLogging`, `validate`, and `clean`.
- replication activity.
- connection accepted events.
- connection closed events.

port

Default: 27017

Specifies a TCP port for the `mongod` or `mongos` (page 676) instance to listen for client connections. UNIX-like systems require root access for ports with numbers lower than 1000.

bind_ip

Default: All interfaces.

Set this option to configure the `mongod` or `mongos` (page 676) process to bind to and listen for connections from applications on this address. You may attach `mongod` or `mongos` (page 676) instances to any interface; however, if you attach the process to a publicly accessible interface, implement proper authentication or firewall restrictions to protect the integrity of your database.

You may concatenate a list of comma separated values to bind `mongod` to multiple IP addresses.

maxConns

Default: depends on system (i.e. ulimit and file descriptor) limits. Unless set, MongoDB will not limit its own connections.

Specifies a value to set the maximum number of simultaneous connections that `mongod` or `mongos` (page 676) will accept. This setting has no effect if it is higher than your operating system's configured maximum connection tracking threshold.

This is particularly useful for `mongos` (page 676) if you have a client that creates a number of collections but allows them to timeout rather than close the collections. When you set `maxConns` (page 622), ensure the value is slightly higher than the size of the connection pool or the total number of connections to prevent erroneous connection spikes from propagating to the members of a *shard* cluster.

Note: You cannot set `maxConns` (page 622) to a value higher than *20000*.

---

objcheck

Default: false

Set to `true` to force `mongod` to validate all requests from clients upon receipt to ensure that invalid *BSON* objects are never inserted into the database. `mongod` does not enable this by default because of the required overhead.

logpath

Default: None. (i.e. `http://docs.mongodb.org/manual/dev/stdout`)

Specify the path to a file name for the log file that will hold all diagnostic logging information.

Unless specified, `mongod` will output all log information to the standard output. Unless `logappend` (page 623) is `true`, the logfile will be overwritten when the process restarts.

Note: Currently, MongoDB will overwrite the contents of the log file if the `logappend` (page 623) is not used. This behavior may change in the future depending on the outcome of SERVER-4499.

---

logappend

Default: false

Set to `true` to add new entries to the end of the logfile rather than overwriting the content of the log when the process restarts.

If this setting is not specified, then MongoDB will overwrite the existing logfile upon start up.

Note: The behavior of the logging system may change in the near future in response to the SERVER-4499 case.

---

syslog

Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with `logpath` (page 622).

Warning: You cannot use `syslog` (page 623) with `logpath` (page 622).

pidfilepath

Default: None.

Specify a file location to hold the "*PID*" or process ID of the `mongod` process. Useful for tracking the `mongod` process in combination with the `fork` (page 623) setting.

Without this option, `mongod` creates no PID file.

keyFile

Default: None.

Specify the path to a key file to store authentication information. This option is only useful for the connection between replica set members.

See Also:

"*Replica Set Security* (page 46)" and "*Replica Set Administration* (page 38)."

nounixsocket

Default: false

Set to `true` to disable listening on the UNIX socket. Unless set to false, `mongod` and `mongos` (page 676) provide a UNIX-socket.

unixSocketPrefix

Default: `http://docs.mongodb.org/manual/tmp`

Specifies a path for the UNIX socket. Unless this option has a value, `mongod` and `mongos` (page 676), create a socket with the `http://docs.mongodb.org/manual/tmp` as a prefix.

fork

Default: false

Set to `true` to enable a *daemon* mode for `mongod` that runs the process in the background.

auth

Default: false

Set to `true` to enable database authentication for users connecting from remote hosts. Configure users via the *mongo shell* (page 591). If no users exist, the localhost interface will continue to have access to the database until the you create the first user.

cpu

Default: false

Set to `true` to force `mongod` to report every four seconds CPU utilization and the amount of time that the processor waits for I/O operations to complete (i.e. I/O wait.) MongoDB writes this data to standard output, or the logfile if using the `logpath` (page 622) option.

dbpath

Default: `http://docs.mongodb.org/manual/data/db/`

Set this value to designate a directory for the `mongod` instance to store its data. Typical locations include: `http://docs.mongodb.org/manual/srv/mongodb`, `http://docs.mongodb.org/manual/var/lib/mongodb` or `http://docs.mongodb.org/manual/opt/mon`

Unless specified, `mongod` will look for data files in the default `http://docs.mongodb.org/manual/data/db` directory. (Windows systems use the `\data\db` directory.) If you installed using a package management system. Check the `http://docs.mongodb.org/manual/etc/mongodb.conf` file provided by your packages to see the configuration of the `dbpath` (page 624).

diaglog

Default: 0

Creates a very verbose, *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the `dbpath` (page 624) directory in a series of files that begin with the string `diaglog` with the time logging was initiated appended as a hex string.

The value of this setting configures the level of verbosity. Possible values, and their impact are as follows.

Value	Setting
0	off. No logging.
1	Log write operations.
2	Log read operations.
3	Log both read and write operations.
7	Log write and some read operations.

You can use the `mongosniff` tool to replay this output for investigation. Given a typical diaglog file, located at `http://docs.mongodb.org/manual/data/db/diaglog.4f76a58c`, you might use a command in the following form to read these files:

```
mongosniff --source DIAGLOG /data/db/diaglog.4f76a58c
```

`diaglog` (page 624) is for internal use and not intended for most users.

Warning: Setting the diagnostic level to `0` will cause `mongod` to stop writing data to the *diagnostic log* file. However, the `mongod` instance will continue to keep the file open, even if it is no longer writing data to the file. If you want to rename, move, or delete the diagnostic log you must cleanly shut down the `mongod` instance before doing so.

directoryperdb
Default: false

Set to `true` to modify the storage pattern of the data directory to store each database's files in a distinct folder. This option will create directories within the `dbpath` (page 624) named for each directory.

Use this option in conjunction with your file system and device configuration so that MongoDB will store data on a number of distinct disk devices to increase write throughput or disk capacity.

journal
Default: (on 64-bit systems) true

Default: (on 32-bit systems) false

Set to true to enable operation journaling to ensure write durability and data consistency.

Set to false to prevent the overhead of journaling in situations where durability is not required. To reduce the impact of the journaling on disk usage, you can leave `journal` (page 625) enabled, and set `smallfiles` (page 627) to true to reduce the size of the data and journal files.

journalCommitInterval
Default: 100

Set this value to specify the maximum amount of time for `mongod` to allow between journal operations. The default value is 100 milliseconds. Lower values increase the durability of the journal, at the possible expense of disk performance.

This option accepts values between 2 and 300 milliseconds.

To force `mongod` to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` pending, `mongod` will reduce `journalCommitInterval` (page 625) to a third of the set value.

ipv6
Default: false

Set to `true` to IPv6 support to allow clients to connect to `mongod` using IPv6 networks. `mongod` disables IPv6 support by default in `mongod` and all utilities.

jsonp
Default: false

Set to `true` to permit *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before setting this option.

noauth
Default: true

Disable authentication. Currently the default. Exists for future compatibility and clarity.

For consistency use the `auth` (page 624) option.

nohttpinterface
Default: false

Set to `true` to disable the HTTP interface. This command will override the `rest` (page 626) and disable the HTTP interface if you specify both. Changed in version 2.1.2: The `nohttpinterface` (page 625) option is not available for `mongos` (page 676) instances before 2.1.2

nojournal
Default: (on 64-bit systems) false

Default: (on 32-bit systems) true

Set `nojournal = true` to disable durability journaling. By default, `mongod` enables journaling in 64-bit versions after v2.0.

noprealloc
Default: false

Set `noprealloc = true` to disable the preallocation of data files. This will shorten the start up time in some cases, but can cause significant performance penalties during normal operations.

noscripting
Default: false

Set `noscripting = true` to disable the scripting engine.

notablescan
Default: false

Set `notablescan = true` to forbid operations that require a table scan.

nssize
Default: 16

Specify this value in megabytes.

Use this setting to control the default size for all newly created namespace files (i.e `.ns`). This option has no impact on the size of existing namespace files.

The default value is 16 megabytes, this provides for effectively 12,000 possible namespace. The maximum size is 2 gigabytes.

profile
Default: 0

Modify this value to changes the level of database profiling, which inserts information about operation performance into output of `mongod` or the log file if specified by `logpath` (page 622). The following levels are available:

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

By default, `mongod` disables profiling. Database profiling can impact database performance because the profiler must record and process all database operations. Enable this option only after careful consideration.

quota
Default: false

Set to `true` to enable a maximum limit for the number data files each database can have. The default quota is 8 data files, when `quota` is true. Adjust the quota size with the with the `quotaFiles` (page 626) setting.

quotaFiles
Default: 8

Modify limit on the number of data files per database. This option requires the `quota` (page 626) setting.

rest
Default: false

Set to `true` to enable a simple *REST* interface.

repair
Default: false

Set to `true` to run a repair routine on all databases following start up. In general you should set this option on the command line and *not* in the *configuration file* (page 163) or in a *control script*.

Use the *`mongod --repair`* (page 585) option to access this functionality.

Note: Because `mongod` rewrites all of the database files during the repair routine, if you do not run `repair` (page 626) under the same user account as `mongod` usually runs, you will need to run `chown` on your database files to correct the permissions before starting `mongod` again.

`repairpath`

Default: `dbpath` (page 624)

Specify the path to the directory containing MongoDB data files, to use in conjunction with the `repair` (page 626) setting or *`mongod --repair`* (page 585) operation. Defaults to the value specified by `dbpath` (page 624).

`slowms`

Default: 100

Specify values in milliseconds.

Sets the threshold for `mongod` to consider a query "slow" for the database profiler. The database logs all slow queries to the log, even when the profiler is not turned on. When the database profiler is on, `mongod` the profiler writes to the `system.profile` collection.

See Also:

"`profile`"

`smallfiles`

Default: false

Set to `true` to modify MongoDB to use a smaller default data file size. Specifically, `smallfiles` (page 627) reduces the initial size for data files and limits them to 512 megabytes. The `smallfiles` (page 627) setting also reduces the size of each *journal* files from 1 gigabyte to 128 megabytes.

Use the `smallfiles` (page 627) setting if you have a large number of databases that each hold a small quantity of data. The `smallfiles` (page 627) setting can lead `mongod` to create many files, which may affect performance for larger databases.

`syncdelay`

Default: 60

This setting controls the maximum number of seconds between flushes of pending writes to disk. While `mongod` is always writing data to disk, this setting controls the maximum guaranteed interval between a successful write operation and the next time the database flushes data to disk.

In many cases, the actual interval between write operations and disk flushes is much shorter than the value

If set to `0`, `mongod` flushes all operations to disk immediately, which may have a significant performance impact. If `journal` (page 625) is `true`, all writes will be durable, by way of the journal within the time specified by `journalCommitInterval` (page 625).

`sysinfo`

Default: false

When set to `true`, `mongod` returns diagnostic system information regarding the page size, the number of physical pages, and the number of available physical pages to standard output.

More typically, run this operation by way of the *`mongod --sysinfo`* (page 586) command. When running with the `sysinfo` (page 627), only `mongod` only outputs the page information and no database process will start.

upgrade

Default: false

When set to `true` this option upgrades the on-disk data format of the files specified by the `dbpath` (page 624) to the latest version, if needed.

This option only affects the operation of `mongod` if the data files are in an old format.

When specified for a `mongos` (page 676) instance, this option updates the meta data format used by the *config database*.

Note: In most cases you should **not** set this value, so you can exercise the most control over your upgrade process. See the MongoDB release notes (on the download page) for more information about the upgrade process.

---

traceExceptions

Default: false

For internal diagnostic use only.

Replication Options

replSet

Default: <none>

Form: <setname>

Use this setting to configure replication with replica sets. Specify a replica set name as an argument to this set. All hosts must have the same set name.

See Also:

"*Replication* (page 31)," "*Replica Set Administration* (page 38)," and "*Replica Set Configuration* (page 661)"

oplogSize

Specifies a maximum size in megabytes for the replication operation log (e.g. *oplog*.) `mongod` creates an oplog based on the maximum amount of space available. For 64-bit systems, the oplog is typically 5% of available disk space.

Once the `mongod` has created the oplog for the first time, changing `oplogSize` (page 628) will not affect the size of the oplog.

fastsync

Default: false

In the context of *replica set* replication, set this option to `true` if you have seeded this replica with a snapshot of the *dbpath* of another member of the set. Otherwise the `mongod` will attempt to perform a full sync.

Warning: If the data is not perfectly synchronized *and* `mongod` starts with `fastsync` (page 628), then the secondary or slave will be permanently out of sync with the primary, which may cause significant consistency problems.

replIndexPrefetch

New in version 2.2. *Default:* `all`

Values: `all`, `none`, and `_id_only`

You must use `replIndexPrefetch` (page 628) in conjunction with `replSet` (page 628).

By default *secondary* members of a *replica set* will load all indexes related to an operation into memory before applying operations from the oplog. You can modify this behavior so that the secondaries will only load the `_id` index. Specify `_id_only` or `none` to prevent the `mongod` from loading *any* index into memory.

Master/Slave Replication

master
Default: false

Set to `true` to configure the current instance to act as *master* instance in a replication configuration.

slave
Default: false

Set to `true` to configure the current instance to act as *slave* instance in a replication configuration.

source
Default: <>

Form: <host>:<port>

Used with the `slave` (page 629) setting to specify the *master* instance from which this *slave* instance will replicate

only
Default: <>

Used with the `slave` (page 629) option, the `only` setting specifies only a single *database* to replicate.

slavedelay
Default: 0

Used with the `slave` (page 629) setting, the `slavedelay` setting configures a "delay" in seconds, for this slave to wait to apply operations from the *master* instance.

autoresync
Default: false

Used with the `slave` (page 629) setting, set `autoresync` to `true` to force the *slave* to automatically resync if the is more than 10 seconds behind the master. This setting may be problematic if the *--oplogSize* (page 586) *oplog* is too small (controlled by the *--oplogSize* (page 586) option.) If the *oplog* not large enough to store the difference in changes between the master's current state and the state of the slave, this instance will forcibly resync itself unnecessarily. When you set the `autoresync` (page 629) option, the slave will not attempt an automatic resync more than once in a ten minute period.

Sharding Cluster Options

configsvr
Default: false

Set this value to `true` to configure this `mongod` instance to operate as the *config database* of a shard cluster. When running with this option, clients will not be able to write data to any database other than `config` and `admin`. The default port for :program:`mongod` with this option is ``27019`` and `mongod` writes all data files to the `http://docs.mongodb.org/manual/configdb` sub-directory of the `dbpath` (page 624) directory.

shardsvr
Default: false

Set this value to `true` to configure this `mongod` instance as a shard in a partitioned cluster. The default port for these instances is `27018`. The only affect of `shardsvr` (page 629) is to change the port number.

noMoveParanoia

Default: false

When set to `true`, `noMoveParanoia` (page 630) disables a "paranoid mode" for data writes for chunk migration operation. See the *chunk migration* (page 138) and `moveChunk` command documentation for more information.

By default `mongod` will save copies of migrated chunks on the "from" server during migrations as "paranoid mode." Setting this option disables this paranoia.

configdb

Default: None.

Format: `<config1>,<config2><:port>,<config3>`

Set this option to specify a configuration database (i.e. *config database*) for the *sharded cluster*. You must specify either 1 configuration server or 3 configuration servers, in a comma separated list.

This setting only affects `mongos` (page 676) processes.

Warning: Never remove a config server from the `configdb` (page 630) parameter, even if the config server or servers are not available, or offline.

test

Default: false

Only runs unit tests and does not start a `mongos` (page 676) instance.

This setting only affects `mongos` (page 676) processes and is for internal testing use only.

chunkSize

Default: 64

The value of this option determines the size of each *chunk* of data distributed around the *sharded cluster*. The default value is 64 megabytes. Larger chunks may lead to an uneven distribution of data, while smaller chunks may lead to frequent and unnecessary migrations. However, in some circumstances it may be necessary to set a different chunk size.

This setting only affects `mongos` (page 676) processes. Furthermore, `chunkSize` (page 630) *only* sets the chunk size when initializing the cluster for the first time. If you modify the run-time option later, the new value will have no effect. See the "*Modify Chunk Size* (page 122)" procedure if you need to change the chunk size on an existing sharded cluster.

localThreshold

New in version 2.2. `localThreshold` (page 630) affects the logic that program:*mongos* uses when selecting *replica set* members to pass reads operations to from clients. Specify a value to `localThreshold` (page 630) in milliseconds. The default value is `15`, which corresponds to the default value in all of the client *drivers* (page 285).

This setting only affects `mongos` (page 676) processes.

When `mongos` (page 676) receives a request that permits reads to *secondary* members, the `mongos` (page 676) will:

- find the member of the set with the lowest ping time.
- construct a list of replica set members that is within a ping time of 15 milliseconds of the nearest suitable member of the set.

If you specify a value for `localThreshold` (page 630), `mongos` (page 676) will construct the list of replica members that are within the latency allowed by this value.

- The `mongos` (page 676) will select a member to read from at random from this list.

The ping time used for a set member compared by the `--localThreshold` setting is a moving average of recent ping times, calculated, at most, every 10 seconds. As a result, some queries may reach members above the threshold until the `mongos` (page 676) recalculates the average.

See the *Member Selection* (page 60) section of the *read preference* (page 55) documentation for more information.

CHAPTER

THIRTYEIGHT

STATUS AND REPORTING

38.1 Server Status Output Index

This document provides a quick overview and example of the the `serverStatus` command. The helper `db.serverStatus()` in the `mongo` shell provides access to this output. For full documentation of the content of this output, see *Server Status Reference* (page 637).

Note: The fields included in this output vary slightly depending on the version of MongoDB, underlying operating system platform, and the kind of node, including `mongos` (page 676), `mongod` or *replica set* member.

The "*Instance Information* (page 637)" section displays information regarding the specific `mongod` and `mongos` (page 676) and its state.

```
{
        "host" : "<hostname>",
        "version" : "<version>",
        "process" : "<mongod|mongos>",
        "pid" : <num>,
        "uptime" : <num>,
        "uptimeMillis" : <num>,
        "uptimeEstimate" : <num>,
        "localTime" : ISODate(""),
```

The "*locks* (page 638)" section reports data that reflect the state and use of both global (i.e. `.`) and database specific locks:

```
"locks" : {
        "." : {
                "timeLockedMicros" : {
                        "R" : <num>,
                        "W" : <num>
                },
                "timeAcquiringMicros" : {
                        "R" : <num>,
                        "W" : <num>
                }
        },
        "admin" : {
                "timeLockedMicros" : {
                        "r" : <num>,
                        "w" : <num>
                },
```

```
                "timeAcquiringMicros" : {
                        "r" : <num>,
                        "w" : <num>
                }
        },
        "local" : {
                "timeLockedMicros" : {
                        "r" : <num>,
                        "w" : <num>
                },
                "timeAcquiringMicros" : {
                        "r" : <num>,
                        "w" : <num>
                }
        },
        "<database>" : {
                "timeLockedMicros" : {
                        "r" : <num>,
                        "w" : <num>
                },
                "timeAcquiringMicros" : {
                        "r" : <num>,
                        "w" : <num>
                }
        }
},
```

The "*globalLock* (page 640)" field reports on MongoDB's global system lock. In most cases the *locks* (page 638) document provides more fine grained data that reflects lock use:

```
"globalLock" : {
        "totalTime" : <num>,
        "lockTime" : <num>,
        "currentQueue" : {
                "total" : <num>,
                "readers" : <num>,
                "writers" : <num>
        },
        "activeClients" : {
                "total" : <num>,
                "readers" : <num>,
                "writers" : <num>
        }
},
```

The "*mem* (page 642)" field reports on MongoDB's current memory use:

```
"mem" : {
        "bits" : <num>,
        "resident" : <num>,
        "virtual" : <num>,
        "supported" : <boolean>,
        "mapped" : <num>,
        "mappedWithJournal" : <num>
},
```

The "*connections* (page 642)" field reports on MongoDB's current memory use by the MongoDB process:

```
"connections" : {
        "current" : <num>,
        "available" : <num>
},
```

The fields in the "*extra_info* (page 643)" document provide platform specific information. The following example block is from a Linux-based system:

```
"extra_info" : {
        "note" : "fields vary by platform",
        "heap_usage_bytes" : <num>,
        "page_faults" : <num>
},
```

The "*indexCounters* (page 643)" document reports on index use:

```
"indexCounters" : {
        "btree" : {
                "accesses" : <num>,
                "hits" : <num>,
                "misses" : <num>,
                "resets" : <num>,
                "missRatio" : <num>
        }
},
```

The "*backgroundFlushing* (page 644)" document reports on the process MongoDB uses to write data to disk:

```
"backgroundFlushing" : {
        "flushes" : <num>,
        "total_ms" : <num>,
        "average_ms" : <num>,
        "last_ms" : <num>,
        "last_finished" : ISODate("")
},
```

The "*cursors* (page 645)" document reports on current cursor use and state:

```
"cursors" : {
        "totalOpen" : <num>,
        "clientCursors_size" : <num>,
        "timedOut" : <num>
},
```

The "*network* (page 645)" document reports on network use and state:

```
"network" : {
        "bytesIn" : <num>,
        "bytesOut" : <num>,
        "numRequests" : <num>
},
```

The "*repl* (page 646)" document reports on the state of replication and the *replica set*. This document only appears for replica sets.

```
"repl" : {
        "setName" : "<string>",
        "ismaster" : <boolean>,
        "secondary" : <boolean>,
        "hosts" : [
```

```
                <hostname>,
                <hostname>,
                <hostname>
        ],
        "primary" : <hostname>,
        "me" : <hostname>
},
```

The "*opcountersRepl* (page 646)" document reports the number of replicated operations:

```
"opcountersRepl" : {
        "insert" : <num>,
        "query" : <num>,
        "update" : <num>,
        "delete" : <num>,
        "getmore" : <num>,
        "command" : <num>
},
```

The "*replNetworkQueue* (page 647)" document holds information regarding the queue that *secondaries* use to poll data from other members of their set:

```
"replNetworkQueue" : {
        "waitTimeMs" : <num>,
        "numElems" : <num>,
        "numBytes" : <num>
},
```

The "*opcounters* (page 648)" document reports the number of operations this MongoDB instance has processed:

```
"opcounters" : {
        "insert" : <num>,
        "query" : <num>,
        "update" : <num>,
        "delete" : <num>,
        "getmore" : <num>,
        "command" : <num>
},
```

The "*asserts* (page 648)" document reports the number of assertions or errors produced by the server:

```
"asserts" : {
        "regular" : <num>,
        "warning" : <num>,
        "msg" : <num>,
        "user" : <num>,
        "rollovers" : <num>
},
```

The "*writeBacksQueued* (page 649)" document reports the number of *writebacks*:

```
"writeBacksQueued" : <num>,
```

The "*dur* (page 649)" document reports on data that reflect this `mongod` instance's journaling-related operations and performance during a journal group commit interval:

```
"dur" : {
        "commits" : <num>,
        "journaledMB" : <num>,
        "writeToDataFilesMB" : <num>,
```

```
        "compression" : <num>,
        "commitsInWriteLock" : <num>,
        "earlyCommits" : <num>,
        "timeMs" : {
                "dt" : <num>,
                "prepLogBuffer" : <num>,
                "writeToJournal" : <num>,
                "writeToDataFiles" : <num>,
                "remapPrivateView" : <num>
        }
},
```

The "*recordStats* (page 650)" document reports data on MongoDB's ability to predict page faults and yield write operations when required data isn't in memory:

```
"recordStats" : {
        "accessesNotInMemory" : <num>,
        "pageFaultExceptionsThrown" : <num>,
        "local" : {
                "accessesNotInMemory" : <num>,
                "pageFaultExceptionsThrown" : <num>
        },
        "<database>" : {
                "accessesNotInMemory" : <num>,
                "pageFaultExceptionsThrown" : <num>
        }
},
```

The final `ok` field holds the return status for the `serverStatus` command:

```
        "ok" : 1
}
```

38.2 Server Status Reference

The `serverStatus` command returns a collection of information that reflects the database's status. These data are useful for diagnosing and assessing the performance of your MongoDB instance. This reference catalogs each datum included in the output of this command and provides context for using this data to more effectively administer your database.

See Also:

Much of the output of `serverStatus` is also displayed dynamically by `mongostat`. See the *mongostat* (page 611) command for more information.

For examples of the `serverStatus` output, see *Server Status Output Index* (page 633).

38.2.1 Instance Information

Example

output of the instance information fields (page 633).

host
The host (page 637) field contains the system's hostname. In Unix/Linux systems, this should be the same as the output of the hostname command.

version
The version (page 638) field contains the version of MongoDB running on the current mongod or mongos (page 676) instance.

process
The process (page 638) field identifies which kind of MongoDB instance is running. Possible values are:

- mongos (page 676)
- mongod

uptime
The value of the uptime (page 638) field corresponds to the number of seconds that the mongos (page 676) or mongod process has been active.

uptimeEstimate
uptimeEstimate (page 638) provides the uptime as calculated from MongoDB's internal course-grained time keeping system.

localTime
The localTime (page 638) value is the current time, according to the server, in UTC specified in an ISODate format.

38.2.2 locks

New in version 2.1.2: All locks (page 671) statuses first appeared in the 2.1.2 development release for the 2.2 series.

Example

output of the locks fields (page 633).

---

locks
The locks (page 671) document contains sub-documents that provides a granular report on MongoDB database-level lock use. All values are of the NumberLong() type.

Generally, fields named:

- R refer to the global read lock,
- W refer to the global write lock,
- r refer to the database specific read lock, and
- w refer to the database specific write lock.

If a document does not have any fields, it means that no locks have existed with this context since the last time the mongod started.

locks..
A field named . holds the first document in locks (page 671) that contains information about the global lock as well as aggregated data regarding lock use in all databases.

locks...**timeLockedMicros**
The locks...timeLockedMicros (page 638) document reports the amount of time in microseconds that a lock has existed in all databases in this mongod instance.

locks...timeLockedMicros.**R**
: The R field reports the amount of time in microseconds that any database has held the global read lock.

locks...timeLockedMicros.**W**
: The W field reports the amount of time in microseconds that any database has held the global write lock.

locks...timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that any database has held the local read lock.

locks...timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that any database has held the local write lock.

locks...**timeAcquiringMicros**
: The locks...timeAcquiringMicros (page 639) document reports the amount of time in microseconds that operations have spent waiting to acquire a lock in all databases in this mongod instance.

locks...timeAcquiringMicros.**R**
: The R field reports the amount of time in microseconds that any database has spent waiting for the global read lock.

locks...timeAcquiringMicros.**W**
: The W field reports the amount of time in microseconds that any database has spent waiting for the global write lock.

locks.**admin**
: The locks.admin (page 639) document contains two sub-documents that report data regarding lock use in the *admin database*.

locks.admin.**timeLockedMicros**
: The locks.admin.timeLockedMicros (page 639) document reports the amount of time in microseconds that locks have existed in the context of the *admin database*.

locks.admin.timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that the *admin database* has held the read lock.

locks.admin.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the *admin database* has held the write lock.

locks.admin.**timeAcquiringMicros**
: The locks.admin.timeAcquiringMicros (page 639) document reports on the amount of field time in microseconds that operations have spent waiting to acquire a lock for the *admin database*.

locks.admin.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the *admin database*.

locks.admin.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the *admin database*.

locks.**local**
: The locks.local (page 639) document contains two sub-documents that report data regarding lock use in the local database. The local database contains a number of instance specific data, including the *oplog* for replication.

locks.local.**timeLockedMicros**
: The locks.local.timeLockedMicros (page 639) document reports on the amount of time in microseconds that locks have existed in the context of the local database.

locks.local.timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that the local database has held the read lock.

locks.local.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the local database has held the write lock.

locks.local.**timeAcquiringMicros**
: The locks.local.timeAcquiringMicros (page 640) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the local database.

locks.local.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the local database.

locks.local.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the local database.

locks.**<database>**
: For each additional database locks (page 671) includes a document that reports on the lock use for this database. The names of these documents reflect the database name itself.

locks.<database>.**timeLockedMicros**
: The locks.<database>.timeLockedMicros (page 640) document reports on the amount of time in microseconds that locks have existed in the context of the <database> database.

locks.<database>.timeLockedMicros.**r**
: The r field reports the amount of time in microseconds that the <database> database has held the read lock.

locks.<database>.timeLockedMicros.**w**
: The w field reports the amount of time in microseconds that the <database> database has held the write lock.

locks.<database>.**timeAcquiringMicros**
: The locks.<database>.timeAcquiringMicros (page 640) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the <database> database.

locks.<database>.timeAcquiringMicros.**r**
: The r field reports the amount of time in microseconds that operations have spent waiting to acquire a read lock on the <database> database.

locks.<database>.timeAcquiringMicros.**w**
: The w field reports the amount of time in microseconds that operations have spent waiting to acquire a write lock on the <database> database.

38.2.3 globalLock

Example

output of the globalLock fields (page 634).

globalLock
: The globalLock (page 640) data structure contains information regarding the database's current lock state, historical lock status, current operation queue, and the number of active clients.

globalLock.**totalTime**
: The value of globalLock.totalTime (page 640) represents the time, in microseconds, since the database last started and creation of the globalLock (page 640). This is roughly equivalent to total server uptime.

globalLock.**lockTime**
: The value of globalLock.lockTime (page 640) represents the time, in microseconds, since the database last started, that the globalLock (page 640) has been *held*.

Consider this value in combination with the value of globalLock.totalTime (page 640). MongoDB aggregates these values in the globalLock.ratio (page 641) value. If the globalLock.ratio (page 641) value is small but globalLock.totalTime (page 640) is high the globalLock (page 640) has typically been held frequently for shorter periods of time, which may be indicative of a more normal use pattern. If the globalLock.lockTime (page 640) is higher and the globalLock.totalTime (page 640) is smaller (relatively,) then fewer operations are responsible for a greater portion of server's use (relatively.)

globalLock.**ratio**
Changed in version 2.2: globalLock.ratio (page 641) was removed. See locks (page 671). The value of globalLock.ratio (page 641) displays the relationship between globalLock.lockTime (page 640) and globalLock.totalTime (page 640).

Low values indicate that operations have held the globalLock (page 640) frequently for shorter periods of time. High values indicate that operations have held globalLock (page 640) infrequently for longer periods of time.

globalLock.currentQueue

globalLock.**currentQueue**
The globalLock.currentQueue (page 641) data structure value provides more granular information concerning the number of operations queued because of a lock.

globalLock.currentQueue.**total**
The value of globalLock.currentQueue.total (page 641) provides a combined total of operations queued waiting for the lock.

A consistently small queue, particularly of shorter operations should cause no concern. Also, consider this value in light of the size of queue waiting for the read lock (e.g. globalLock.currentQueue.readers (page 641)) and write-lock (e.g. globalLock.currentQueue.writers (page 641)) individually.

globalLock.currentQueue.**readers**
The value of globalLock.currentQueue.readers (page 641) is the number of operations that are currently queued and waiting for the read-lock. A consistently small read-queue, particularly of shorter operations should cause no concern.

globalLock.currentQueue.**writers**
The value of globalLock.currentQueue.writers (page 641) is the number of operations that are currently queued and waiting for the write-lock. A consistently small write-queue, particularly of shorter operations is no cause for concern.

globalLock.activeClients

globalLock.**activeClients**
The globalLock.activeClients (page 641) data structure provides more granular information about the number of connected clients and the operation types (e.g. read or write) performed by these clients.

Use this data to provide context for the *currentQueue* (page 641) data.

globalLock.activeClients.**total**
The value of globalLock.activeClients.total (page 641) is the total number of active client connections to the database. This combines clients that are performing read operations (e.g. globalLock.activeClients.readers (page 641)) and clients that are performing write operations (e.g. globalLock.activeClients.writers (page 641)).

globalLock.activeClients.**readers**
The value of globalLock.activeClients.readers (page 641) contains a count of the active client connections performing read operations.

globalLock.activeClients.**writers**
The value of globalLock.activeClients.writers (page 641) contains a count of active client connections performing write operations.

38.2.4 mem

Example

output of the memory fields (page 634).

mem
The mem data structure holds information regarding the target system architecture of mongod and current memory use.

mem.**bits**
The value of mem.bits (page 642) is either 64 or 32, depending on which target architecture specified during the mongod compilation process. In most instances this is 64, and this value does not change over time.

mem.**resident**
The value of mem.resident (page 642) is roughly equivalent to the amount of RAM, in bytes, currently used by the database process. In normal use this value tends to grow. In dedicated database servers this number tends to approach the total amount of system memory.

mem.**virtual**
mem.virtual (page 642) displays the quantity, in bytes, of virtual memory used by the mongod process. In typical deployments this value is slightly larger than mem.mapped (page 642). If this value is significantly (i.e. gigabytes) larger than mem.mapped (page 642), this could indicate a memory leak.

With *journaling* enabled, the value of mem.virtual (page 642) is twice the value of mem.mapped (page 642).

mem.**supported**
mem.supported (page 642) is true when the underlying system supports extended memory information. If this value is false and the system does not support extended memory information, then other mem values may not be accessible to the database server.

mem.**mapped**
The value of mem.mapped (page 642) provides the amount of mapped memory by the database. Because MongoDB uses memory-mapped files, this value is likely to be to be roughly equivalent to the total size of your database or databases.

38.2.5 connections

Example

output of the connections fields (page 634).

connections
The connections sub document data regarding the current connection status and availability of the database server. Use these values to asses the current load and capacity requirements of the server.

connections.**current**
The value of connections.current (page 642) corresponds to the number of connections to the

database server from clients. This number includes the current shell session. Consider the value of `connections.available` (page 643) to add more context to this datum.

This figure will include the current shell connection as well as any inter-node connections to support a *replica set* or *sharded cluster*.

`connections.`**`available`**
`connections.available` (page 643) provides a count of the number of unused available connections that the database can provide. Consider this value in combination with the value of `connections.current` (page 642) to understand the connection load on the database, and the *Linux ulimit Settings* (page 189) document for more information about system thresholds on available connections.

38.2.6 extra_info

Example

output of the extra_info fields (page 635).

`extra_info`
The `extra_info` (page 643) data structure holds data collected by the `mongod` instance about the underlying system. Your system may only report a subset of these fields.

`extra_info.`**`note`**
The field `extra_info.note` (page 643) reports that the data in this structure depend on the underlying platform, and has the text: "fields vary by platform."

`extra_info.`**`heap_usage_bytes`**
The `extra_info.heap_usage_bytes` (page 643) field is only available on Unix/Linux systems, and reports the total size in bytes of heap space used by the database process.

`extra_info.`**`page_faults`**
The `extra_info.page_faults` (page 643) field is only available on Unix/Linux systems, and reports the total number of page faults that require disk operations. Page faults refer to operations that require the database server to access data which isn't available in active memory. The `page_fault` (page 643) counter may increase dramatically during moments of poor performance and may correlate with limited memory environments and larger data sets. Limited and sporadic page faults do not necessarily indicate an issue.

38.2.7 indexCounters

Example

output of the indexCounters fields (page 635).

`indexCounters`
Changed in version 2.2: Previously, data in the `indexCounters` (page 643) document reported sampled data, and were only useful in relative comparison to each other, because they could not reflect absolute index use. In 2.2 and later, these data reflect actual index use. The `indexCounters` (page 643) data structure reports information regarding the state and use of indexes in MongoDB.

`indexCounters.`**`btree`**
The `indexCounters.btree` (page 643) data structure contains data regarding MongoDB's *btree* indexes.

indexCounters.btree.**accesses**
: indexCounters.btree.accesses (page 643) reports the number of times that operations have accessed indexes. This value is the combination of the indexCounters.btree.hits (page 644) and indexCounters.btree.misses (page 644). Higher values indicate that your database has indexes and that queries are taking advantage of these indexes. If this number does not grow over time, this might indicate that your indexes do not effectively support your use.

indexCounters.btree.**hits**
: The indexCounters.btree.hits (page 644) value reflects the number of times that an index has been accessed and mongod is able to return the index from memory.

  A higher value indicates effective index use. indexCounters.btree.hits (page 644) values that represent a greater proportion of the indexCounters.btree.accesses (page 643) value, tend to indicate more effective index configuration.

indexCounters.btree.**misses**
: The indexCounters.btree.misses (page 644) value represents the number of times that an operation attempted to access an index that was not in memory. These "misses," do not indicate a failed query or operation, but rather an inefficient use of the index. Lower values in this field indicate better index use and likely overall performance as well.

indexCounters.btree.**resets**
: The indexCounters.btree.resets (page 644) value reflects the number of times that the index counters have been reset since the database last restarted. Typically this value is 0, but use this value to provide context for the data specified by other indexCounters (page 643) values.

indexCounters.btree.**missRatio**
: The indexCounters.btree.missRatio (page 644) value is the ratio of indexCounters.btree.hits (page 644) to indexCounters.btree.misses (page 644) misses. This value is typically 0 or approaching 0.

38.2.8 backgroundFlushing

Example

output of the backgroundFlushing fields (page 635).

backgroundFlushing
: mongod periodically flushes writes to disk. In the default configuration, this happens every 60 seconds. The backgroundFlushing (page 644) data structure contains data regarding these operations. Consider these values if you have concerns about write performance and *journaling* (page 649).

backgroundFlushing.**flushes**
: backgroundFlushing.flushes (page 644) is a counter that collects the number of times the database has flushed all writes to disk. This value will grow as database runs for longer periods of time.

backgroundFlushing.**total_ms**
: The backgroundFlushing.total_ms (page 644) value provides the total number of milliseconds (ms) that the mongod processes have spent writing (i.e. flushing) data to disk. Because this is an absolute value, consider the value of backgroundFlushing.flushes (page 644) and backgroundFlushing.average_ms (page 644) to provide better context for this datum.

backgroundFlushing.**average_ms**
: The backgroundFlushing.average_ms (page 644) value describes the relationship between the number of flushes and the total amount of time that the database has spent writing data to disk. The larger

backgroundFlushing.flushes (page 644) is, the more likely this value is likely to represent a "normal," time; however, abnormal data can skew this value.

Use the backgroundFlushing.last_ms (page 645) to ensure that a high average is not skewed by transient historical issue or a random write distribution.

backgroundFlushing.**last_ms**
The value of the backgroundFlushing.last_ms (page 645) field is the amount of time, in milliseconds, that the last flush operation took to complete. Use this value to verify that the current performance of the server and is in line with the historical data provided by backgroundFlushing.average_ms (page 644) and backgroundFlushing.total_ms (page 644).

backgroundFlushing.**last_finished**
The backgroundFlushing.last_finished (page 645) field provides a timestamp of the last completed flush operation in the *ISODate* format. If this value is more than a few minutes old relative to your server's current time and accounting for differences in time zone, restarting the database may result in some data loss.

Also consider ongoing operations that might skew this value by routinely block write operations.

38.2.9 cursors

Example

output of the cursors (page 635) fields.

cursors
The cursors data structure contains data regarding cursor state and use.

cursors.**totalOpen**
cursors.totalOpen (page 645) provides the number of cursors that MongoDB is maintaining for clients. Because MongoDB exhausts unused cursors, typically this value small or zero. However, if there is a queue, stale tailable cursors, or a large number of operations this value may rise.

cursors.**clientCursors_size**
Deprecated since version 1.x: See cursors.totalOpen (page 645) for this datum.

cursors.**timedOut**
cursors.timedOut (page 645) provides a counter of the total number of cursors that have timed out since the server process started. If this number is large or growing at a regular rate, this may indicate an application error.

38.2.10 network

Example

output of the network fields (page 635).

network
The network data structure contains data regarding MongoDB's network use.

network.**bytesIn**
The value of the network.bytesIn (page 645) field reflects the amount of network traffic, in bytes, received *by* this database. Use this value to ensure that network traffic sent to the mongod process is consistent with expectations and overall inter-application traffic.

network.**bytesOut**
: The value of the network.bytesOut (page 645) field reflects the amount of network traffic, in bytes, sent *from* this database. Use this value to ensure that network traffic sent by the mongod process is consistent with expectations and overall inter-application traffic.

network.**numRequests**
: The network.numRequests (page 646) field is a counter of the total number of distinct requests that the server has received. Use this value to provide context for the network.bytesIn (page 645) and network.bytesOut (page 645) values to ensure that MongoDB's network utilization is consistent with expectations and application use.

38.2.11 repl

Example

output of the repl fields (page 635).

repl
: The repl data structure contains status information for MongoDB's replication (i.e. "replica set") configuration. These values only appear when the current host has replication enabled.

  See *Replication Fundamentals* (page 33) for more information on replication.

repl.**setName**
: The repl.setName (page 646) field contains a string with the name of the current replica set. This value reflects the *--replSet* (page 586) command line argument, or replSet (page 628) value in the configuration file.

  See *Replication Fundamentals* (page 33) for more information on replication.

repl.**ismaster**
: The value of the repl.ismaster (page 646) field is either true or false and reflects whether the current node is the master or primary node in the replica set.

  See *Replication Fundamentals* (page 33) for more information on replication.

repl.**secondary**
: The value of the repl.secondary (page 646) field is either true or false and reflects whether the current node is a secondary node in the replica set.

  See *Replication Fundamentals* (page 33) for more information on replication.

repl.**hosts**
: repl.hosts (page 646) is an array that lists the other nodes in the current replica set. Each member of the replica set appears in the form of hostname:port.

  See *Replication Fundamentals* (page 33) for more information on replication.

38.2.12 opcountersRepl

Example

output of the opcountersRepl fields (page 636).

opcountersRepl
The opcountersRepl (page 646) data structure, similar to the opcounters data structure, provides an overview of database replication operations by type and makes it possible to analyze the load on the replica in more granular manner. These values only appear when the current host has replication enabled.

These values will differ from the opcounters values because of how MongoDB serializes operations during replication. See *Replication Fundamentals* (page 33) for more information on replication.

These numbers will grow over time in response to database use. Analyze these values over time to track database utilization.

opcountersRepl.**insert**
opcountersRepl.insert (page 647) provides a counter of the total number of replicated insert operations since the mongod instance last started.

opcountersRepl.**query**
opcountersRepl.query (page 647) provides a counter of the total number of replicated queries since the mongod instance last started.

opcountersRepl.**update**
opcountersRepl.update (page 647) provides a counter of the total number of replicated update operations since the mongod instance last started.

opcountersRepl.**delete**
opcountersRepl.delete (page 647) provides a counter of the total number of replicated delete operations since the mongod instance last started.

opcountersRepl.**getmore**
opcountersRepl.getmore (page 647) provides a counter of the total number of "getmore" operations since the mongod instance last started. This counter can be high even if the query count is low. Secondary nodes send getMore operations as part of the replication process.

opcountersRepl.**command**
opcountersRepl.command (page 647) provides a counter of the total number of replicated commands issued to the database since the mongod instance last started.

38.2.13 replNetworkQueue

New in version 2.1.2.

Example

output of the replNetworkQueue fields (page 636).

---

replNetworkQueue
The replNetworkQueue (page 647) document reports on the network replication buffer, which permits replication operations to happen in the background. This feature is internal.

This document only appears on *secondary* members of *replica sets*.

replNetworkQueue.**waitTimeMs**
replNetworkQueue.waitTimeMs (page 647) reports the amount of time that a *secondary* waits to add operations to network queue. This value is cumulative.

replNetworkQueue.**numElems**
replNetworkQueue.numElems (page 647) reports the number of operations stored in the queue.

replNetworkQueue.**numBytes**
replNetworkQueue.numBytes (page 647) reports the total size of the network replication queue.

38.2.14 opcounters

Example

output of the opcounters fields (page 636).

opcounters

The `opcounters` data structure provides an overview of database operations by type and makes it possible to analyze the load on the database in more granular manner.

These numbers will grow over time and in response to database use. Analyze these values over time to track database utilization.

`opcounters.`**insert**

`opcounters.insert` (page 648) provides a counter of the total number of insert operations since the `mongod` instance last started.

`opcounters.`**query**

`opcounters.query` (page 648) provides a counter of the total number of queries since the `mongod` instance last started.

`opcounters.`**update**

`opcounters.update` (page 648) provides a counter of the total number of update operations since the `mongod` instance last started.

`opcounters.`**delete**

`opcounters.delete` (page 648) provides a counter of the total number of delete operations since the `mongod` instance last started.

`opcounters.`**getmore**

`opcounters.getmore` (page 648) provides a counter of the total number of "getmore" operations since the `mongod` instance last started. This counter can be high even if the query count is low. Secondary nodes send `getMore` operations as part of the replication process.

`opcounters.`**command**

`opcounters.command` (page 648) provides a counter of the total number of commands issued to the database since the `mongod` instance last started.

38.2.15 asserts

Example

output of the asserts fields (page 636).

asserts

The `asserts` data structure provides an account of the number of asserts on the database. While assert errors are typically uncommon, if there are non-zero values for the `asserts`, you should check the log file for the `mongod` process for more information. In many cases these errors are trivial, but are worth investigating.

`asserts.`**regular**

The `asserts.regular` (page 648) counter tracks the number of regular assertions raised since the server process started. Check the log file for more information about these messages.

`asserts.`**warning**

The `asserts.warning` (page 648) counter tracks the number of warnings raised since the server process started. Check the log file for more information about these warnings.

`asserts.msg`
: The `asserts.msg` (page 648) counter tracks the number of message assertions raised since the server process started. Check the log file for more information about these messages.

`asserts.user`
: The `asserts.users` counter reports the number of "user asserts" that have occurred since the last time the server process started. These are errors that user may generate, such as out of disk space or duplicate key. You can prevent these assertions by fixing a problem with your application or deployment. Check the MongoDB log for more information.

`asserts.rollovers`
: The `asserts.rollovers` (page 649) counter displays the number of times that the rollover counters have rolled over since the last time the server process started. The counters will rollover to zero after 2^{30} assertions. Use this value to provide context to the other values in the `asserts` data structure.

38.2.16 writeBacksQueued

Example

output of the writeBacksQueued fields (page 636).

writeBacksQueued
: The value of `writeBacksQueued` is `true` when there are operations from a `mongos` (page 676) instance queued for retrying. Typically this option is false.

See Also:

writeBacks

38.2.17 dur

New in version 1.8.

Journaling

Example

output of the journaling fields (page 636).

dur
: The `dur` (for "durability") document contains data regarding the `mongod`'s journaling-related operations and performance. `mongod` must be running with journaling for these data to appear in the output of "`serverStatus`".

Note: The data values are **not** cumulative but are reset on a regular basis as determined by the journal group commit interval. This interval is ~100 milliseconds (ms) by default (or 30ms if the journal file is on the same file system as your data files) and is cut by 1/3 when there is a `getLastError` command pending. The interval is configurable using the `--journalCommitInterval` option.

See Also:

"Journaling" for more information about journaling operations.

dur.**commits**
The dur.commits (page 649) provides the number of transactions written to the *journal* during the last journal group commit interval.

dur.**journaledMB**
The dur.journaledMB (page 650) provides the amount of data in megabytes (MB) written to *journal* during the last journal group commit interval.

dur.**writeToDataFilesMB**
The dur.writeToDataFilesMB (page 650) provides the amount of data in megabytes (MB) written from *journal* to the data files during the last journal group commit interval.

dur.**compression**
New in version 2.0. The dur.compression (page 650) represents the compression ratio of the data written to the *journal*:

```
( journaled_size_of_data / uncompressed_size_of_data )
```

dur.**commitsInWriteLock**
The dur.commitsInWriteLock (page 650) provides a count of the commits that occurred while a write lock was held. Commits in a write lock indicate a MongoDB node under a heavy write load and call for further diagnosis.

dur.**earlyCommits**
The dur.earlyCommits (page 650) value reflects the number of times MongoDB requested a commit before the scheduled journal group commit interval. Use this value to ensure that your journal group commit interval is not too long for your deployment.

dur.**timeMS**
The dur.timeMS (page 650) document provides information about the performance of the mongod instance during the various phases of journaling in the last journal group commit interval.

dur.timeMS.**dt**
The dur.timeMS.dt (page 650) value provides, in milliseconds, the amount of time over which MongoDB collected the dur.timeMS (page 650) data. Use this field to provide context to the other dur.timeMS (page 650) field values.

dur.timeMS.**prepLogBuffer**
The dur.timeMS.prepLogBuffer (page 650) value provides, in milliseconds, the amount of time spent preparing to write to the journal. Smaller values indicate better journal performance.

dur.timeMS.**writeToJournal**
The dur.timeMS.writeToJournal (page 650) value provides, in milliseconds, the amount of time spent actually writing to the journal. File system speeds and device interfaces can affect performance.

dur.timeMS.**writeToDataFiles**
The dur.timeMS.writeToDataFiles (page 650) value provides, in milliseconds, the amount of time spent writing to data files after journaling. File system speeds and device interfaces can affect performance.

dur.timeMS.**remapPrivateView**
The dur.timeMS.remapPrivateView (page 650) value provides, in milliseconds, the amount of time spent remapping copy-on-write memory mapped views. Smaller values indicate better journal performance.

38.2.18 recordStats

Example

output of the recordStats (page 637) fields.

recordStats
The recordStats (page 651) document provides fine grained reporting on page faults on a per database level.

recordStats.**accessesNotInMemory**
recordStats.accessesNotInMemory (page 651) reflects the number of times mongod needed to access a memory page that was *not* resident in memory for *all* databases managed by this mongod instance.

recordStats.**pageFaultExceptionsThrown**
recordStats.pageFaultExceptionsThrown (page 651) reflects the number of page fault exceptions thrown by mongod when accessing data for *all* databases managed by this mongod instance.

recordStats.local.**accessesNotInMemory**
recordStats.local.accessesNotInMemory (page 651) reflects the number of times mongod needed to access a memory page that was *not* resident in memory for the local database.

recordStats.local.**pageFaultExceptionsThrown**
recordStats.local.pageFaultExceptionsThrown (page 651) reflects the number of page fault exceptions thrown by mongod when accessing data for the local database.

recordStats.admin.**accessesNotInMemory**
recordStats.admin.accessesNotInMemory (page 651) reflects the number of times mongod needed to access a memory page that was *not* resident in memory for the *admin database*.

recordStats.admin.**pageFaultExceptionsThrown**
recordStats.admin.pageFaultExceptionsThrown (page 651) reflects the number of page fault exceptions thrown by mongod when accessing data for the *admin database*.

recordStats.<database>.**accessesNotInMemory**
recordStats.<database>.accessesNotInMemory (page 651) reflects the number of times mongod needed to access a memory page that was *not* resident in memory for the <database> database.

recordStats.<database>.**pageFaultExceptionsThrown**
recordStats.<database>.pageFaultExceptionsThrown (page 651) reflects the number of page fault exceptions thrown by mongod when accessing data for the <database> database.

38.3 Database Statistics Reference

38.3.1 Synopsis

MongoDB can report data that reflects the current state of the "active" database. In this context "database," refers to a single MongoDB database. To run dbStats issue this command in the shell:

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

The mongo shell provides the helper function db.stats(). Use the following form:

```
db.stats()
```

The above commands are equivalent. Without any arguments, db.stats() returns values in bytes. To convert the returned values to kilobytes, use the scale argument:

```
db.stats(1024)
```

Or:

```
db.runCommand( { dbStats: 1, scale: 1024 } )
```

Note: Because scaling rounds values to whole number, scaling may return unlikely or unexpected results.

The above commands are equivalent. See the `dbStats` *database command* and the `db.stats()` helper for the `mongo` shell for additional information.

38.3.2 Fields

db
Contains the name of the database.

collections
Contains a count of the number of collections in that database.

objects
Contains a count of the number of objects (i.e. *documents*) in the database across all collections.

avgObjSize
The average size of each object. The `scale` argument affects this value. This is the `dataSize` divided by the number of objects.

dataSize
The total size of the data held in this database including the *padding factor*. The `scale` argument affects this value. The `dataSize` will not decrease when *documents* shrink, but will decrease when you remove documents.

storageSize
The total amount of space allocated to collections in this database for *document* storage. The `scale` argument affects this value. The `storageSize` (page 652) does not decrease as you remove or shrink documents.

numExtents
Contains a count of the number of extents in the database across all collections.

indexes
Contains a count of the total number of indexes across all collections in the database.

indexSize
The total size of all indexes created on this database. The `scale` arguments affects this value.

fileSize
The total size of the data files that hold the database. This value includes preallocated space and the *padding factor*. The value of `fileSize` (page 652) only reflects the size of the data files for the database and not the namespace file.

The `scale` argument affects this value.

nsSizeMB
The total size of the *namespace* files (i.e. that end with `.ns`) for this database. You cannot change the size of the namespace file after creating a database, but you can change the default size for all new namespace files with the `nssize` (page 626) runtime option.

See Also:

The `nssize` (page 626) option, and *Maximum Namespace File Size* (page 679)

38.4 Collection Statistics Reference

38.4.1 Synopsis

To fetch collection statistics, call the `db.collection.stats()` method on a collection object in the `mongo` shell:

```
db.collection.stats()
```

You may also use the literal command format:

```
db.runCommand( { collStats: "collection" } )
```

Replace `collection` in both examples with the name of the collection you want statistics for. By default, the return values will appear in terms of bytes. You can, however, enter a `scale` argument. For example, you can convert the return values to kilobytes like so:

```
db.collection.stats(1024)
```

Or:

```
db.runCommand( { collStats: "collection", scale: 1024 } )
```

Note: The `scale` argument rounds values to whole numbers. This can produce unpredictable and unexpected results in some situations.

See Also:

The documentation of the "`collStats`" command and the "`db.collection.stats()`," method in the `mongo` shell.

38.4.2 Example Document

The output of `db.collection.stats()` resembles the following:

```
{
     "ns" : "<database>.<collection>",
     "count" : <number>,
     "size" : <number>,
     "avgObjSize" : <number>,
     "storageSize" : <number>,
     "numExtents" : <number>,
     "nindexes" : <number>,
     "lastExtentSize" : <number>,
     "paddingFactor" : <number>,
     "systemFlags" : <bit>,
     "userFlags" : <bit>,
     "totalIndexSize" : <number>,
     "indexSizes" : {
             "_id_" : <number>,
             "a_1" : <number>
     },
     "ok" : 1
}
```

38.4.3 Fields

ns
The namespace of the current collection, which follows the format `[database].[collection]`.

count
The number of objects or documents in this collection.

size
The size of the data stored in this collection. This value does not include the size of any indexes associated with the collection, which the `totalIndexSize` (page 654) field reports.

The `scale` argument affects this value.

avgObjSize
The average size of an object in the collection. The `scale` argument affects this value.

storageSize
The total amount of storage allocated to this collection for *document* storage. The `scale` argument affects this value. The `storageSize` (page 652) does not decrease as you remove or shrink documents.

numExtents
The total number of contiguously allocated data file regions.

nindexes
The number of indexes on the collection. All collections have at least one index on the *_id* field. Changed in version 2.2: Before 2.2, capped collections did not necessarily have an index on the `_id` field, and some capped collections created with pre-2.2 versions of `mongod` may not have an `_id` index.

lastExtentSize
The size of the last extent allocated. The `scale` argument affects this value.

paddingFactor
The amount of space added to the end of each document at insert time. The document padding provides a small amount of extra space on disk to allow a document to grow slightly without needing to move the document. `mongod` automatically calculates this padding factor

flags
Changed in version 2.2: Removed in version 2.2 and replaced with the `userFlags` (page 654) and `systemFlags` (page 654) fields. Indicates the number of flags on the current collection. In version 2.0, the only flag notes the existence of an *index* on the *_id* field.

systemFlags
New in version 2.2. Reports the flags on this collection that reflect internal server options. Typically this value is `1` and reflects the existence of an *index* on the `_id` field.

userFlags
New in version 2.2. Reports the flags on this collection set by the user. In version 2.2 the only user flag is `usePowerOf2Sizes`.

See `collMod` for more information on setting user flags and *usePowerOf2Sizes*.

totalIndexSize
The total size of all indexes. The `scale` argument affects this value.

indexSizes
This field specifies the key and size of every existing index on the collection. The `scale` argument affects this value.

38.5 Collection Validation Data

38.5.1 Synopsis

The collection validation command checks all of the structures within a name space for correctness and returns a *document* containing information regarding the on-disk representation of the collection.

Warning: The `validate` process may consume significant system resources and impede application performance because it must scan all data in the collection.

Run the validation command in the `mongo` shell using the following form to validate a collection named `people`:

```
db.people.validate()
```

Alternatively you can use the command prototype and the `db.runCommand()` shell helper in the following form:

```
db.runCommand( { validate: "people", full: true } )
db.people.validate(true)
```

See Also:

"`validate`" and "`validate()`."

38.5.2 Values

ns
The full namespace name of the collection. Namespaces include the database name and the collection name in the form `database.collection`.

firstExtent
The disk location of the first extent in the collection. The value of this field also includes the namespace.

lastExtent
The disk location of the last extent in the collection. The value of this field also includes the namespace.

extentCount
The number of extents in the collection.

extents
`validate` returns one instance of this document for every extent in the collection. This sub-document is only returned when you specify the `full` option to the command.

`extents.`**loc**
The disk location for the beginning of this extent.

`extents.`**xnext**
The disk location for the extent following this one. "null" if this is the end of the linked list of extents.

`extents.`**xprev**
The disk location for the extent preceding this one. "null" if this is the head of the linked list of extents.

`extents.`**nsdiag**
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

`extents.`**size**
The number of bytes in this extent.

`extents.firstRecord`
The disk location of the first record in this extent.

`extents.lastRecord`
The disk location of the last record in this extent.

datasize
The number of bytes in all data records. This value does not include deleted records, nor does it include extent headers, nor record headers, nor space in a file unallocated to any extent. `datasize` (page 656) includes record *padding*.

nrecords
The number of *documents* in the collection.

lastExtentSize
The size of the last new extent created in this collection. This value determines the size of the *next* extent created.

padding
A floating point value between 1 and 2.

When MongoDB creates a new record it uses the *padding factor* to determine how much additional space to add to the record. The padding factor is automatically adjusted by mongo when it notices that update operations are triggering record moves.

firstExtentDetails
The size of the first extent created in this collection. This data is similar to the data provided by the `extents` (page 655) sub-document; however, the data reflects only the first extent in the collection and is always returned.

`firstExtentDetails.loc`
The disk location for the beginning of this extent.

`firstExtentDetails.xnext`
The disk location for the extent following this one. "null" if this is the end of the linked list of extents, which should only be the case if there is only one extent.

`firstExtentDetails.xprev`
The disk location for the extent preceding this one. This should always be "null."

`firstExtentDetails.nsdiag`
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

`firstExtentDetails.size`
The number of bytes in this extent.

`firstExtentDetails.firstRecord`
The disk location of the first record in this extent.

`firstExtentDetails.lastRecord`
The disk location of the last record in this extent.

objectsFound
The number of records actually encountered in a scan of the collection. This field should have the same value as the `nrecords` (page 656) field.

invalidObjects
The number of records containing BSON documents that do not pass a validation check.

Note: This field is only included in the validation output when you specify the `full` option.

---

bytesWithHeaders
This is similar to datasize, except that `bytesWithHeaders` (page 656) includes the record headers. In version 2.0, record headers are 16 bytes per document.

Note: This field is only included in the validation output when you specify the `full` option.

---

bytesWithoutHeaders
`bytesWithoutHeaders` (page 657) returns data collected from a scan of all records. The value should be the same as `datasize` (page 656).

Note: This field is only included in the validation output when you specify the `full` option.

---

deletedCount
The number of deleted or "free" records in the collection.

deletedSize
The size of all deleted or "free" records in the collection.

nIndexes
The number of indexes on the data in the collection.

keysPerIndex
A document containing a field for each index, named after the index's name, that contains the number of keys, or documents referenced, included in the index.

valid
Boolean. `true`, unless `validate` determines that an aspect of the collection is not valid. When `false`, see the `errors` (page 657) field for more information.

errors
Typically empty; however, if the collection is not valid (i.e `valid` (page 657) is false,) this field will contain a message describing the validation error.

ok
Set to `1` when the command succeeds. If the command fails the `ok` (page 657) field has a value of `0`.

38.6 Connection Pool Statistics Reference

38.6.1 Synopsis

`mongos` (page 676) instances maintain a pool of connections for interacting with constituent members of the *sharded cluster*. Additionally, `mongod` instances maintain connection with other shards in the cluster for migrations. The `connPoolStats` command returns statistics regarding these connections between the `mongos` (page 676) and `mongod` instances or between the `mongod` instances in a shard cluster.

Note: `connPoolStats` only returns meaningful results for `mongos` (page 676) instances and for `mongod` instances in sharded clusters.

38.6.2 Output

hosts
The sub-documents of the `hosts` (page 657) *document* report connections between the `mongos` (page 676) or

mongod instance and each component mongod of the *sharded cluster*.

hosts.[host].**available**
hosts.[host].available (page 658) reports the total number of connections that the mongos (page 676) or mongod could use to connect to this mongod.

hosts.[host].**created**
hosts.[host].created (page 658) reports the number of connections that this mongos (page 676) or mongod has ever created for this host.

replicaSets
replicaSets (page 658) is a *document* that contains *replica set* information for the *sharded cluster*.

replicaSets.**shard**
The replicaSets.shard (page 658) *document* reports on each *shard* within the *sharded cluster*

replicaSets.[shard].**host**
The replicaSets.[shard].host (page 658) field holds an array of *document* that reports on each host within the *shard* in the *replica set*.

These values derive from the *replica set status* (page 659) values.

replicaSets.[shard].host[n].**addr**
replicaSets.[shard].host[n].addr (page 658) reports the address for the host in the *sharded cluster* in the format of "[hostname]:[port]".

replicaSets.[shard].host[n].**ok**
replicaSets.[shard].host[n].ok (page 658) reports false when:
- the mongos (page 676) or mongod cannot connect to instance.
- the mongos (page 676) or mongod received a connection exception or error.

This field is for internal use.

replicaSets.[shard].host[n].**ismaster**
replicaSets.[shard].host[n].ismaster (page 658) reports true if this replicaSets.[shard].host (page 658) is the *primary* member of the *replica set*.

replicaSets.[shard].host[n].**hidden**
replicaSets.[shard].host[n].hidden (page 658) reports true if this replicaSets.[shard].host (page 658) is a *hidden member* of the *replica set*.

replicaSets.[shard].host[n].**secondary**
replicaSets.[shard].host[n].secondary (page 658) reports true if this replicaSets.[shard].host (page 658) is a *secondary* member of the *replica set*.

replicaSets.[shard].host[n].**pingTimeMillis**
replicaSets.[shard].host[n].pingTimeMillis (page 658) reports the ping time in milliseconds from the mongos (page 676) or mongod to this replicaSets.[shard].host (page 658).

replicaSets.[shard].host[n].**tags**
New in version 2.2. replicaSets.[shard].host[n].tags (page 658) reports the members[n].tags (page 663), if this member of the set has tags configured.

replicaSets.[shard].**master**
replicaSets.[shard].master (page 658) reports the ordinal identifier of the host in the replicaSets.[shard].host (page 658) array that is the *primary* of the *replica set*.

replicaSets.[shard].**nextSlave**
Deprecated since version 2.2. replicaSets.[shard].nextSlave (page 658) reports the *secondary* member that the mongos (page 676) will use to service the next request for this *replica set*.

createdByType

createdByType (page 658) *document* reports the number of each type of connection that mongos (page 676) or mongod has created in all connection pools.

mongos (page 676) connect to mongod instances using one of three types of connections. The following sub-document reports the total number of connections by type.

createdByType.**master**

createdByType.master (page 659) reports the total number of connections to the *primary* member in each *cluster*.

createdByType.**set**

createdByType.set (page 659) reports the total number of connections to a *replica set* member.

createdByType.**sync**

createdByType.sync (page 659) reports the total number of *config database* connections.

totalAvailable

totalAvailable (page 659) reports the running total of connections from the mongos (page 676) or mongod to all mongod instances in the *sharded cluster* available for use. This value does not reflect those connections that

totalCreated

totalCreated (page 659) reports the total number of connections ever created from the mongos (page 676) or mongod to all mongod instances in the *sharded cluster*.

numDBClientConnection

numDBClientConnection (page 659) reports the total number of connections from the mongos (page 676) or mongod to all of the mongod instances in the *sharded cluster*.

numAScopedConnection

numAScopedConnection (page 659) reports the number of exception safe connections created from mongos (page 676) or mongod to all mongod in the *sharded cluster*. The mongos (page 676) or mongod releases these connections after receiving a socket exception from the mongod.

38.7 Replica Set Status Reference

The replSetGetStatus provides an overview of the current status of a *replica set*. Issue the following command against the *admin database*, in the mongo shell:

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

You can also use the following helper in the mongo shell to access this functionality

```
rs.status()
```

The value specified (e.g 1 above,) does not impact 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.

Note: The mongod must have replication enabled and be a member of a replica set for the for replSetGetStatus to return successfully.

See Also:

"rs.status()" shell helper function, "*Replication* (page 31)".

38.7.1 Fields

`rs.status.set`
The `set` value is the name of the replica set, configured in the `replSet` (page 628) setting. This is the same value as `_id` (page 662) in `rs.conf()`.

`rs.status.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 `members.lastHeartbeat` (page 661) to find the operational lag between the current host and the other hosts in the set.

`rs.status.myState`
The value of `myState` (page 660) reflects state of the current replica set member. An integer between `0` and `10` represents the state of the member. These integers map to states, as described in the following table:

Number	State
0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering (initial syncing, post-rollback, stale members)
4	Fatal error
5	Starting up, phase 2 (forking threads)
6	Unknown state (the set has never connected to the member)
7	Arbiter
8	Down
9	Rollback
10	Removed

`rs.status.members`
The `members` field holds an array that contains a document for every member in the replica set. See the "*Member Statuses* (page 660)" for an overview of the values included in these documents.

`rs.status.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.

38.7.2 Member Statuses

`members.name`
The `name` field holds the name of the server.

`members.self`
The `self` field is only included in the document for the current `mongod` instance in the members array. It's value is `true`.

`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.

`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`.)

`members.state`
The value of the `members.state` (page 660) reflects state of this replica set member. An integer between `0` and `10` represents the state of the member. These integers map to states, as described in the following table:

Number	State
0	Starting up, phase 1 (parsing configuration)
1	Primary
2	Secondary
3	Recovering (initial syncing, post-rollback, stale members)
4	Fatal error
5	Starting up, phase 2 (forking threads)
6	Unknown state (the set has never connected to the member)
7	Arbiter
8	Down
9	Rollback
10	Removed

`members.stateStr`
A string that describes `members.state` (page 660).

`members.uptime`
The `members.uptime` (page 661) 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.

`members.optime`
A document that contains information regarding the last operation from the operation log that this member has applied.

`members.optime.t`
A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

`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.

`members.optimeDate`
An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from `members.lastHeartbeat` (page 661) 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.

`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` (page 660) field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` data.

`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.

38.8 Replica Set Configuration

38.8.1 Synopsis

This reference provides an overview of all possible 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.

38.8.2 Configuration Variables

`rs.conf._id`
Type: string

Value: <setname>

An `_id` field holding the name of the replica set. This reflects the set name configured with `replSet` (page 628) or *`mongod --replSet`* (page 586).

`rs.conf.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 662) field in the replica set configuration document is a zero-indexed array.

`members[n]._id`
Type: ordinal

Provides the zero-indexed identifier of every member in the replica set.

`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:** `members[n].host` (page 662) 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.

`members[n].arbiterOnly`
Optional.

Type: boolean

Default: false

Identifies an arbiter. For arbiters, this value is `true`, and is automatically configured by `rs.addArb()`".

`members[n].buildIndexes`
Optional.

Type: boolean

Default: true

Determines whether the `mongod` builds *indexes* on this member. Do not set to `false` if a replica set *can* become a master, or if any clients ever issue queries against this instance.

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 will, *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 `members[n].buildIndexes` (page 662) field after adding the member to the set.
> Furthermore, other secondaries cannot synchronize off of replica set members where `members[n].buildIndexes` (page 662) is true.

`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 40)"

`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 elections from members when another eligible member has a higher absolute priority value. Changing the balance of priority in a replica set will cause an election.

A `members[n].priority` (page 663) of `0` makes it impossible for a member to become primary.

See Also:

"*Replica Set Member Priority* (page 34)" and "*Replica Set Elections* (page 34)."

`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* (page 54) to allow configurable data center awareness.

Use in conjunction with `settings.getLastErrorModes` (page 664) and `settings.getLastErrorDefaults` (page 664) and `db.getLastError()` (i.e. `getLastError`.)

`members[n].slaveDelay`
Optional.

Type: Integer. (seconds.)

Default: 0

Describes the number of seconds "behind" the master that this replica set member should "lag." Use this option to create *delayed members* (page 40), that maintain a copy of the data that reflects the state of the data set some

amount of time (specified in seconds.) Typically these members help protect against human error, and provide some measure of insurance against the unforeseen consequences of changes and updates.

`members[n].votes`
Optional.

Type: Integer

Default: 1

Controls the number of votes a server has in a *replica set election* (page 34). 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, use this setting to add additional non-voting members with a `members[n].votes` (page 664) value of `0`.

For most deployments and most members, use the default value, `1`, for `members[n].votes` (page 664).

`settings`
Optional.

Type: *MongoDB Document*

The setting document holds two optional fields, which affect the available *write concern* options and default configurations.

`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.

`settings.getLastErrorModes`
Optional.

Type: *MongoDB Document*

Defines the names and combination of `tags` (page 663) for use by the application layer to guarantee *write concern* to database using the `getLastError` command to provide *data-center awareness*.

38.8.3 Example Document

The following document provides a representation of a replica set configuration document. Angle brackets (e.g. < and >) enclose all optional fields.

```
{
  _id : <setname>,
  members: [
    {
      _id : <ordinal>,
      <host : <hostname<:port>>,>
      <arbiterOnly : <boolean>,>
      <buildIndexes : <bool>,>
      <hidden : <boolean>,>
      <priority: <priority>,>
      <tags: { <document> },>
      <slaveDelay : <number>,>
      <votes : <number>>
    }
    , ...
```

```
  ],
  <settings: {
    <getLastErrorDefaults : <lasterrdefaults>,>
    <getLastErrorModes : <modes>>
  }>
}
```

38.8.4 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"
            }
     ]
}
```

And the following reconfiguration operation:

```
cfg = rs.conf()
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
rs.reconfig(cfg)
```

This operation begins by saving the current replica set configuration to the local variable `cfg` using the `rs.conf()` method. Then it adds priority values to the *document* where the `members[n]._id` (page 662) field has a value of `0`, `1`, or `2`. 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 662). 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 command can force the current primary to step down and causes an election in some situations. When the primary steps down, all clients will disconnect. This is by design. While this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods.

38.8.5 Tag Sets

Tag sets provide custom and configurable *write concern* (page 54) and *read preferences* (page 55). This section will outline the process for specifying tags for a replica set, for more information see the full documentation of the behavior of *tags sets write concern* (page 54) and *tag sets for read preference* (page 58).

Configure tag sets by adding fields and values to the document stored in the `members[n].tags` (page 663). Consider the following example:

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"
                }
        ]
}
```

You could add the 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",
                                "group": "production"
                        }
                },
                {
                        "_id" : 1,
                        "host" : "mongodb1.example.net:27017",
                        "tags" : {
                                "dc": "east",
                                "group": "reporting"
                        }
                },
                {
                        "_id" : 2,
                        "host" : "mongodb2.example.net:27017",
                        "tags" : {
                                "group": "production"
                        }
                }
        ]
}
```

38.9 Replication Info Reference

The `db.getReplicationInfo()` provides current status of the current replica status, using data polled from the "*oplog*". Consider the values of this output when diagnosing issues with replication.

See Also:

"*Replication Fundamentals* (page 33)" for more information on replication.

38.9.1 All Nodes

The following fields are present in the output of `db.getReplicationInfo()` for both *primary* and *secondary* nodes.

logSizeMB
Returns the total size of the *oplog* in megabytes. This refers to the total amount of space allocated to the oplog rather than the current size of operations stored in the oplog.

usedMB
Returns the total amount of space used by the *oplog* in megabytes. This refers to the total amount of space currently used by operations stored in the oplog rather than the total amount of space allocated.

38.9.2 Primary Nodes

The following fields appear in the output of `db.getReplicationInfo()` for *primary* nodes.

errmsg
Returns the last error status.

oplogMainRowCount
Returns a counter of the number of items or rows (i.e. *documents*) in the *oplog*.

38.9.3 Secondary Nodes

The following fields appear in the output of `db.getReplicationInfo()` for *secondary* nodes.

timeDiff
Returns the difference between the first and last operation in the *oplog*, represented in seconds.

timeDiffHours
Returns the difference between the first and last operation in the *oplog*, rounded and represented in hours.

tFirst
Returns a time stamp for the first (i.e. earliest) operation in the *oplog*. Compare this value to the last write operation issued against the server.

tLast
Returns a time stamp for the last (i.e. latest) operation in the *oplog*. Compare this value to the last write operation issued against the server.

now
Returns a time stamp reflecting the current time. The shell process generates this value, and the datum may differ slightly from the server time if you're connecting from a remote host as a result. Equivalent to `Date()`.

38.10 Current Operation Reporting

Changed in version 2.2.

38.10.1 Example Output

The `db.currentOp()` helper in the `mongo` shell reports on the current operations running on the `mongod` instance. The operation returns the `inprog` array, which contains a document for each in progress operation. Consider the following example output:

```
{
  "inprog": [
             {
                        "opid" : 3434473,
                        "active" : <boolean>,
                        "secs_running" : 0,
                        "op" : "<operation>",
                        "ns" : "<database>.<collection>",
                        "query" : {
                        },
                        "client" : "<host>:<outgoing>",
                        "desc" : "conn57683",
                        "threadId" : "0x7f04a637b700",
                        "connectionId" : 57683,
                        "locks" : {
                                "^" : "w",
                                "^local" : "W",
                                "^<database>" : "W"
                        },
                        "waitingForLock" : false,
                        "msg": "<string>"
                        "numYields" : 0,
                        "progress" : {
                                "done" : <number>,
                                "total" : <number>
                        }
                        "lockStats" : {
                                "timeLockedMicros" : {
                                        "R" : NumberLong(),
                                        "W" : NumberLong(),
                                        "r" : NumberLong(),
                                        "w" : NumberLong()
                                },
                                "timeAcquiringMicros" : {
                                        "R" : NumberLong(),
                                        "W" : NumberLong(),
                                        "r" : NumberLong(),
                                        "w" : NumberLong()
                                }
                        }
             },
            ]
}
```

Optional

You may specify the `true` argument to `db.currentOp()` to return a more verbose output including idle connections and system operations. For example:

```
db.currentOp(true)
```

Furthermore, active operations (i.e. where `active` (page 670) is `true`) will return additional fields.

38.10.2 Operations

You can use the `db.killOp()` in conjunction with the `opid` (page 670) field to terminate a currently running operation. Consider the following JavaScript operations for the `mongo` shell that you can use to filter the output of identify specific types of operations:

- Return all pending write operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.waitingForLock && d.lockType != "read")
        printjson(d)
     })
```

- Return the active write operation:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "write")
        printjson(d)
     })
```

- Return all active read operations:

```
db.currentOp().inprog.forEach(
   function(d){
     if(d.active && d.lockType == "read")
        printjson(d)
     })
```

38.10.3 Output Reference

opid
Holds an identifier for the operation. You can pass this value to `db.killOp()` in the `mongo` shell to terminate the operation.

active
A boolean value, that is `true` if the operation is currently running or `false` if the operation is queued and waiting for a lock to run.

secs_running
The duration of the operation in seconds. MongoDB calculates this value by subtracting the current time from the start time of the operation.

op
A string that identifies the type of operation. The possible values are:

- `insert`
- `query`
- `update`
- `remove`
- `getmore`
- `command`

ns
The *namespace* the operation targets. MongoDB forms namespaces using the name of the *database* and the name of the *collection*.

query
A document containing the current operation's query. The document is empty for operations that do not have queries: `getmore`, `insert`, and `command`.

client
The IP address (or hostname) and the ephemeral port of the client connection where the operation originates. If your `inprog` array has operations from many different clients, use this string to relate operations to clients.

For some commands, including `findAndModify` and `db.eval()`, the client will be `0.0.0.0:0`, rather than an actual client.

desc
A description of the client. This string includes the `connectionId` (page 671).

threadId
An identifier for the thread that services the operation and its connection.

connectionId
An identifier for the connection where the operation originated.

locks
New in version 2.2. The `locks` (page 671) document reports on the kinds of locks the operation currently holds. The following kinds of locks are possible:

`locks.^`
`locks.^` (page 671) reports on the global lock state for the `mongod` instance. The operation must hold this for some global phases of an operation.

`locks.^local`
`locks.^` (page 671) reports on the lock for the `local` database. MongoDB uses the `local` database for a number of operations, but the most frequent use of the `local` database is for the *oplog* used in replication.

`locks.^<database>`
`locks.^` reports on the lock state for the database that this operation targets.

`locks` (page 671) replaces `lockType` in earlier versions.

lockType
Changed in version 2.2: The `locks` (page 671) replaced the `lockType` (page 671) field in 2.2. Identifies the type of lock the operation currently holds. The possible values are:

- `read`
- `write`

waitingForLock
Returns a boolean value. `waitingForLock` (page 671) is `true` if the operation is waiting for a lock and `false` if the operation has the required lock.

msg
The `msg` (page 671) provides a message that describes the status and progress of the operation. In the case of indexing or mapReduce operations, the field reports the completion percentage.

progress
Reports on the progress of mapReduce or indexing operations. The `progress` (page 671) fields corresponds to the completion percentage in the `msg` (page 671) field. The `progress` (page 671) specifies the following information:

done
Reports the number completed.

total
Reports the total number.

killed
Returns `true` if `mongod` instance is in the process of killing the operation.

numYields
`numYields` (page 672) is a counter that reports the number of times the operation has 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 quickly while MongoDB reads in data for the yielding operation.

lockStats
New in version 2.2. The `lockStats` (page 672) document reflects the amount of time the operation has spent both acquiring and holding locks. `lockStats` (page 672) reports data on a per-lock type, with the following possible lock types:

- `R` represents the global read lock,
- `W` represents the global write lock,
- `r` represents the database specific read lock, and
- `w` represents the database specific write lock.

timeLockedMicros
The `timeLockedMicros` (page 672) document reports the amount of time the operation has spent holding a specific lock.

`timeLockedMicros.R`
Reports the amount of time in microseconds the operation has held the global read lock.

`timeLockedMicros.W`
Reports the amount of time in microseconds the operation has held the global write lock.

`timeLockedMicros.r`
Reports the amount of time in microseconds the operation has held the database specific read lock.

`timeLockedMicros.w`
Reports the amount of time in microseconds the operation has held the database specific write lock.

timeAcquiringMicros
The `timeAcquiringMicros` (page 672) document reports the amount of time the operation has spent *waiting* to acquire a specific lock.

`timeAcquiringMicros.R`
Reports the mount of time in microseconds the operation has waited for the global read lock.

`timeAcquiringMicros.W`
Reports the mount of time in microseconds the operation has waited for the global write lock.

`timeAcquiringMicros.r`
Reports the mount of time in microseconds the operation has waited for the database specific read lock.

`timeAcquiringMicros.w`
: Reports the mount of time in microseconds the operation has waited for the database specific write lock.

38.11 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` (page 676) 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 678) 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` (page 586) option to upgrade the database to the version supported by this `mongod` instance.

5
: Returned by `mongod` if a `moveChunk` operation fails to confirm a commit.

12
: Returned by the `mongod.exe` process on Windows when it receives a Control-C, Close, Break or Shutdown event.

14
: Returned by MongoDB applications which encounter an unrecoverable error, an uncaught exception or uncaught signal. The system exits without performing a clean shut down.

20
: *Message:* `ERROR: wsastartup failed <reason>`

Returned by MongoDB applications on Windows following an error in the WSAStartup function.

Message: `NT Service Error`

Returned by MongoDB applications for Windows due to failures installing, starting or removing the NT Service for the application.

45
: Returned when a MongoDB application cannot open a file or cannot obtain a lock on a file.

47
: MongoDB applications exit cleanly following a large clock skew (32768 milliseconds) event.

48
: `mongod` exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the `--port` (page 582) run-time option.

49

Returned by `mongod.exe` or `mongos.exe` on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100

Returned by `mongod` when the process throws an uncaught exception.

CHAPTER

THIRTYNINE

INTERNAL METADATA

39.1 Config Database Contents

The `config` database supports *sharded cluster* operation. See the *Sharding* (page 105) section of this manual for full documentation of sharded clusters.

To access a the `config` database, connect to a `mongos` (page 676) instance in a sharded cluster, and issue the following command:

```
use config
```

You can return a list of the databases, by issuing the following command:

```
show collections
```

39.1.1 Collections

chunks

The `chunks` (page 675) collection stores a document for each chunk in 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.

collections

The `collections` (page 652) 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 652) 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")
}
```

databases

The databases (page 676) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. databases (page 676) 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" }
```

lockpings

The lockpings (page 676) collection keeps track of the active components in the sharded cluster. Given a cluster with a mongos (page 676) running on example.com:30000, the document in the lockpings (page 676) collection would resemble:

```
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```

locks

The locks (page 671) collection stores a distributed lock. This ensures that only one mongos (page 676) instance can perform administrative tasks on the cluster at once. The mongos (page 676) 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 (page 676) 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.

mongos

The mongos (page 676) collection stores a document for each mongos (page 676) instance affiliated with the cluster. mongos (page 676) instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the mongos (page 676) is active. The ping field shows the time of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the mongos (page 676) running on example.com:30000.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait
```

settings

The settings (page 664) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see *Modify Chunk Size* (page 122).

•Balancer status. To change status, see *Disable the Balancer* (page 126).

The following is an example `settings` collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }
```

shards

The `shards` (page 677) 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" }
```

version

The `version` (page 638) collection holds the current metadata version number. This collection contains only one document:

To access the `version` (page 638) 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` collection contains metadata for all indexes in the database for information on indexes, see *Indexes* (page 229).

39.2 Local Database

39.2.1 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.

When running with authentication (i.e. `auth` (page 624)), authenticating against the `local` database is equivalent to authenticating against the `admin` database. This authentication gives access to all databases.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` database contains the following collections used for replication:

39.2.2 Collections on Replica Set Members

`local.system.replset`

`local.system.replset` (page 677) 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 677) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSize` (page 628) setting. To resize the oplog after replica set initiation, use the *Change the Size of the Oplog* (page 85) procedure. For additional information, see the *Oplog Internals* (page 62) topic in this document and the *Oplog* (page 36) topic in the *Replication Fundamentals* (page 33) document.

`local.replset.`**`minvalid`**
: This contains an object used internally by replica sets to track sync status.

39.2.3 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.

39.3 System Collections

39.3.1 Synopsis

MongoDB stores system information in collections that use the `<database>.system.*` *namespace*, which MongoDB reserves for internal use. Do not create collections that begin with `system.`.

MongoDB also stores some additional instance-local metadata in the *local database* (page 677), specifically for replication purposes.

39.3.2 Collections

System collections include these collections stored directly in the database:

`<database>.system.`**`namespaces`**
: The `<database>.system.namespaces` (page 678) 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 678) 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 678) collection stores database profiling information. For information on profiling, see *Database Profiling* (page 174).

`<database>.system.`**`users`**
: The `<database>.system.users` (page 678) collection stores credentials for users who have access to the database. For more information on this collection, see *Authentication* (page 210).

CHAPTER

FORTY

GENERAL REFERENCE

40.1 MongoDB Limits and Thresholds

40.1.1 Synopsis

This document provides a collection of hard and soft limitations of the MongoDB system.

40.1.2 Limits

BSON Documents

BSON Document Size
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 285) for more information about GridFS.

Nested Depth for BSON Documents
Changed in version 2.2. MongoDB supports no more than 100 levels of nesting for *BSON documents*.

Namespaces

Namespace Length
Each namespace, including database and collection name, must be shorter than 123 bytes.

Number of Namespaces
The limitation on the number of namespaces is the size of the namespace file divided by 628.

A 16 megabyte namespace file can support approximately 24,000 namespaces.

Size of Namespace File
Namespace files can be no larger than 2 gigabytes.

By default namespace files are 16 megabytes. You can configure the size using the `nssize` (page 626).

Indexes

Index Size
Indexed items can be *no larger* than 1024 bytes. This value is the indexed content (i.e. the field value, or compound field value.)

Number of Indexes per Collection
A single collection can have *no more* than 64 indexes.

Index Name Length
The names of indexes, including their namespace (i.e database and collection name) cannot be longer than 128 characters. The default index name is the concatenation of the field names and index directions.

You can explicitly specify a name to `createIndex` or the `db.collection.ensureIndex()` helper if the default index name is too long.

Unique Indexes in Sharded Collections
MongoDB does not support unique indexes across shards, except when the unique index contains the full shard key as a prefix of the index. In these situations MongoDB will enforce uniqueness across the full key, not a single field.

See Also:

Enforce Unique Keys for Sharded Collections (page 147) for an alternate approach.

Number of Indexed Fields in a Compound Index:
There can be no more than 31 fields in a compound index.

Replica Sets

Number of Members of a Replica Set
Replica sets can ho more than 12 members.

Number of Voting Members of a Replica Set
Only 7 members of a replica set can have votes at any given time. See can vote *Non-Voting Members* (page 42) for more information

Operations

Sorted Documents
MongoDB will only return sorted results on fields without an index *if* the sort operation uses less than 32 megabytes of memory.

Operations Unavailable in Sharded Environments
The `group` does not work with sharding. Use `mapreduce` or `aggregate` instead.

`db.eval()` is incompatible with sharded collections. You may use `db.eval()` with un-sharded collections in a shard cluster.

`$where` does not permit references to the `db` object from the `$where` function. This is uncommon in un-sharded collections.

The `$atomic` update modifier does not work in sharded environments.

`$snapshot` (page 464) queries do not work in sharded environments.

Naming Restrictions

Restrictions on Database Names
The dot (i.e. `.`) character is not permissible in database names.

Database names are case sensitive even if the underlying file system is case insensitive. Changed in version 2.2: For MongoDB instances running on Windows. In 2.2 the following characters are not permissible in database names:

```
/\. "*<>:|?
```

See *Restrictions on Database Names for Windows* (page 698) for more information.

Restriction on Collection Names
New in version 2.2. Collection names should begin with an underscore or a letter character, and *cannot*:

- contain the `$`.
- be an empty string (e.g. `""`).
- contain the null character.
- begin with the `system.` prefix. (Reserved for internal use.)

See *Are there any restrictions on the names of Collections?* (page 421) and *Restrictions on Collection Names* (page 697) for more information.

Restrictions on Field Names
Field names cannot contain dots (i.e. `.`) or dollar signs (i.e. `$`.) See *Dollar Sign Operator Escaping* (page 419) for an alternate approach.

40.2 Glossary

$cmd A virtual *collection* that exposes *MongoDB*'s *database commands*.

_id A field containing a unique ID, typically a BSON *ObjectId*. If not specified, this value is automatically assigned upon the creation of a new document. You can think of the `_id` as the document's *primary key*.

accumulator An *expression* in the *aggregation framework* that maintains state between documents in the *aggregation pipeline*. See: `$group` (page 473) for a list of accumulator operations.

admin database A privileged database named `admin`. Users must have access to this database to run certain administrative commands. See *administrative commands* for more information and *admin-commands* for a list of these commands.

aggregation Any of a variety of operations that reduce and summarize large sets of data. SQL's `GROUP` and MongoDB's map-reduce are two examples of aggregation functions.

aggregation framework The MongoDB aggregation framework provides a means to calculate aggregate values without having to use *map-reduce*.

See Also:

Aggregation Framework (page 253).

arbiter A member of a *replica set* that exists solely to vote in *elections*. Arbiter nodes do not replicate data.

See Also:

Delayed Nodes (page 40)

balancer An internal MongoDB process that runs in the context of a *sharded cluster* and manages the splitting and migration of *chunks*. Administrators must disable the balancer for all maintenance operations on a sharded cluster.

box MongoDB's *geospatial* indexes and querying system allow you to to build queries around rectangles on two-dimensional coordinate systems. These queries use the `$box` operator to define a shape using the lower-left and the upper-right coordinates.

BSON A serialization format used to store documents and make remote procedure calls in MongoDB. "BSON" is a a portmanteau of the words "binary" and "JSON". Think of BSON as a binary representation of JSON (JavaScript Object Notation) documents. For a detailed spec, see bsonspec.org.

See Also:

The *Data Type Fidelity* (page 177) section.

BSON types The set of types supported by the *BSON* serialization format. The following types are available:

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

btree A data structure used by most database management systems for to store indexes. MongoDB uses b-trees for its indexes.

CAP Theorem Given three properties of computing systems, consistency, availability, and partition tolerance, a distributed computing system can provide any two of these features, but never all three.

capped collection A fixed-sized *collection*. Once they reach their fixed size, capped collections automatically overwrite their oldest entries. MongoDB's *oplog* replication mechanism depends on capped collections. Developers may also use capped collections in their applications.

See Also:

The Capped Collections wiki page.

checksum A calculated value used to ensure data integrity. The *md5* algorithm is sometimes used as a checksum.

chunk In the context of a *sharded cluster*, a chunk is a contiguous range of *shard key* values assigned to a particular *shard*. By default, chunks are 64 megabytes or less. When they grow beyond the configured chunk size, a `mongos` (page 676) splits the chunk into two chunks.

circle MongoDB's *geospatial* indexes and querying system allow you to build queries around circles on two-dimensional coordinate systems. These queries use the `$circle` operator to define circle using the center

and the radius of the circle.

client The application layer that uses a database for data persistence and storage. *Drivers* provide the interface level between the application layer and the database server.

cluster A set of `mongod` instances running in conjunction to increase database availability and performance. See *sharding* and *replication* for more information on the two different approaches to clustering with MongoDB.

collection A namespace within a database for containing *documents*. Collections do not enforce a schema, but they are otherwise mostly analogous to *RDBMS* tables.

compound index An *index* consisting of two or more keys. See *Indexing Overview* (page 231) for more information.

config database One of three `mongod` instances that store all of the metadata associated with a *sharded cluster*.

control script A simple shell script, typically located in the `http://docs.mongodb.org/manual/etc/rc.d` or `http://docs.mongodb.org/manual/etc/init.d` directory and used by the system's initialization process to start, restart and stop a *daemon* process.

control script A script used by a UNIX-like operating system to start, stop, or restart a *daemon* process. On most systems, you can find these scripts in the `http://docs.mongodb.org/manual/etc/init.d/` or `http://docs.mongodb.org/manual/etc/rc.d/` directories.

CRUD Create, read, update, and delete. The fundamental operations of any database.

CSV A text-based data format consisting of comma-separated values. This format is commonly used to exchange database between relational databases, since the format is well-suited to tabular data. You can import CSV files using `mongoimport`.

cursor In MongoDB, a cursor is a pointer to the result set of a *query*, that clients can iterate through to retrieve results. By default, cursors will timeout after 10 minutes of inactivity.

daemon The conventional name for a background, non-interactive process.

data-center awareness A property that allows clients to address nodes in a system to based upon their location.

Replica sets implement data-center awareness using *tagging*.

See Also:

`members[n].tags` (page 663) and *Tag Sets* (page 666) for more information about tagging and replica sets.

database A physical container for *collections*. Each database gets its own set of files on the file system. A single MongoDB server typically servers multiple databases.

database command Any MongoDB operation other than an insert, update, remove, or query. MongoDB exposes commands as queries against the special *$cmd* collection. For example, the implementation of `count` for MongoDB is a command.

See Also:

`http://docs.mongodb.org/manual/reference/commands` for a full list of database commands in MongoDB

database profiler A tool that, when enabled, keeps a record on all long-running operations in a database's `system.profile` collection. The profiler is most often used to diagnose slow queries.

See Also:

Monitoring Database Systems (page 174).

dbpath Refers to the location of MongoDB's data file storage. The default `dbpath` (page 624) is `http://docs.mongodb.org/manual/data/db`. Other common data paths include `http://docs.mongodb.org/manual/srv/mongodb` and `http://docs.mongodb.org/manual/var/lib/mongodb`.

See Also:

dbpath (page 624) or --dbpath (page 583).

delayed member A member of a *replica set* that cannot become primary and applies operations at a specified delay. This delay is useful for protecting data from human error (i.e. unintentionally deleted databases) or updates that have unforeseen effects on the production database.

See Also:

Delayed Members (page 40)

diagnostic log mongod can create a verbose log of operations with the *mongod --diaglog* (page 583) option or through the diagLogging command. The mongod creates this log in the directory specified to *mongod --dbpath* (page 583). The name of the is diaglog.<time in hex>, where "<time-in-hex>" reflects the initiation time of logging as a hexadecimal string.

> **Warning:** Setting the diagnostic level to 0 will cause mongod to stop writing data to the *diagnostic log* file. However, the mongod instance will continue to keep the file open, even if it is no longer writing data to the file. If you want to rename, move, or delete the diagnostic log you must cleanly shut down the mongod instance before doing so.

See Also:

mongod --diaglog (page 583), diaglog (page 624), and diagLogging.

document A record in a MongoDB collection, and the basic unit of data in MongoDB. Documents are analogous to JSON objects, but exist in the database in a more type-rich format known as *BSON*.

draining The process of removing or "shedding" *chunks* from one *shard* to another. Administrators must drain shards before removing them from the cluster.

See Also:

removeShard, *sharding*.

driver A client implementing the communication protocol required for talking to a server. The MongoDB drivers provide language-idiomatic methods for interfacing with MongoDB.

See Also:

Drivers (page 285)

election In the context of *replica sets*, an election is the process by which members of a replica set select primary nodes on startup and in the event of failures.

See Also:

Replica Set Elections (page 34) and *priority*.

eventual consistency A property of a distributed system allowing changes to the system to propagate gradually. In a database system, this means that readable nodes are not required to reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondary nodes have *eventual consistency*.

expression In the context of the *aggregation framework*, expressions are the stateless transformations that operate on the data that passes through the *pipeline*.

See Also:

Aggregation Framework (page 253).

failover The process that allows one of the *secondary* nodes in a *replica set* to become *primary* in the event of a failure.

See Also:

Replica Set Failover (page 34).

field A name-value pair in a *document*. Documents have zero or more fields. Fields are analogous to columns in relational databases.

firewall A system level networking filter that restricts access based on, among other things, IP address. Firewalls form part of effective network security strategy.

fsync A system call that flushes all dirty, in-memory pages to disk. MongoDB calls `fsync()` on its database files at least every 60 seconds.

Geohash A value is a binary representation of the location on a coordinate grid.

geospatial Data that relates to geographical location. In MongoDB, you may index or store geospatial data according to geographical parameters and reference specific coordinates in queries.

GridFS A convention for storing large files in a MongoDB database. All of the official MongoDB drivers support this convention, as does the `mongofiles` program.

See Also:

mongofiles (page 619).

haystack index In the context of *geospatial* queries, haystack indexes enhance searches by creating "bucket" of objects grouped by a second criterion. For example, you might want want all geographical searches to also include the type of location being searched for. In this case, you can create a haystack index that includes a document's position and type:

```
db.places.ensureIndex( { position: "geoHaystack", type: 1 } )
```

You can then query on position and type:

```
db.places.find( { position: [34.2, 33.3], type: "restaurant" } )
```

hidden member A member of a *replica set* that cannot become primary and is not advertised as part of the set in the *database command* `isMaster`, which prevents it from receiving read-only queries depending on *read preference*.

See Also:

Hidden Member (page 40), `isMaster`, `db.isMaster`, and `members[n].hidden` (page 663).

idempotent When calling an idempotent operation on a value or state, the operation only affects the value once. Thus, the operation can safely run multiple times without unwanted side effects. In the context of MongoDB, *oplog* entries must be idempotent to support initial synchronization and recovery from certain failure situations. Thus, MongoDB can safely apply oplog entries more than once without any ill effects.

index A data structure that optimizes queries. See *Indexing Overview* (page 231) for more information.

IPv6 A revision to the IP (Internet Protocol) standard that provides a significantly larger address space to more effectively support the number of hosts on the contemporary Internet.

ISODate The international date format used by `mongo`. to display dates. E.g. `YYYY-MM-DD HH:MM.SS.milis`.

JavaScript A popular scripting language original designed for web browsers. The MongoDB shell and certain server-side functions use a JavaScript interpreter.

journal A sequential, binary transaction used to bring the database into a consistent state in the event of a hard shutdown. MongoDB enables journaling by default for 64-bit builds of MongoDB version 2.0 and newer. Journal files are pre-allocated and will exist as three 1GB file in the data directory. To make journal files smaller, use `smallfiles` (page 627).

When enabled, MongoDB writes data first to the journal and after to the core data files. MongoDB commits to the journal every 100ms and this is configurable using the `journalCommitInterval` (page 625) runtime option.

To force `mongod` to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` pending, `mongod` will reduce `journalCommitInterval` (page 625) to a third of the set value.

See Also:

The Journaling wiki page.

JSON JavaScript Object Notation. A human-readable, plain text format for expressing structured data with support in many programming languages.

JSON document A *JSON* document is a collection of fields and values in a structured format. The following is a sample *JSON document* with two fields:

```
{ name: "MongoDB",
  type: "database" }
```

JSONP *JSON* with Padding. Refers to a method of injecting JSON into applications. Presents potential security concerns.

LVM Logical volume manager. LVM is a program that abstracts disk images from physical devices, and provides a number of raw disk manipulation and snapshot capabilities useful for system management.

map-reduce A data and processing and aggregation paradigm consisting of a "map" phase that selects data, and a "reduce" phase that transforms the data. In MongoDB, you can run arbitrary aggregations over data using map-reduce.

See Also:

The Map Reduce wiki page for more information regarding MongoDB's map-reduce implementation, and *Aggregation Framework* (page 253) for another approach to data aggregation in MongoDB.

master In conventional master/*slave* replication, the master database receives all writes. The *slave* instances replicate from the master instance in real time.

md5 `md5` is a hashing algorithm used to efficiently provide reproducible unique strings to identify and *checksum* data. MongoDB uses md5 to identify chunks of data for *GridFS*.

MIME "Multipurpose Internet Mail Extensions." A standard set of type and encoding definitions used to declare the encoding and type of data in multiple data storage, transmission, and email contexts.

mongo The MongoDB Shell. `mongo` connects to `mongod` and `mongos` (page 676) instances, allowing administration, management, and testing. `mongo` has a JavaScript interface.

See Also:

mongo (page 591) and `http://docs.mongodb.org/manual/reference/javascript`.

mongod The program implementing the MongoDB database server. This server typically runs as a *daemon*.

See Also:

mongod (page 581).

MongoDB The document-based database server described in this manual.

mongos The routing and load balancing process that acts an interface between an application and a MongoDB *sharded cluster*.

See Also:

mongos (page 588).

multi-master replication A *replication* method where multiple database instances can accept write operations to the same data set at any time. Multi-master replication exchanges increased concurrency and availability for a relaxed consistency semantic. MongoDB ensures consistency and, therefore, does not provide multi-master replication.

namespace A canonical name for a collection or index in MongoDB. Namespaces consist of a concatenation of the database and collection or index name, like so: `[database-name].[collection-or-index-name]`. All documents belong to a namespace.

natural order The order in which a database stores documents on disk. Typically, the order of documents on disks reflects insertion order, *except* when documents move internal because of document growth due to update operations. However, *Capped collections* guarantee that insertion order and natural order are identical.

When you execute `find()` with no parameters, the database returns documents in forward natural order. When you execute `find()` and include `sort()` with a parameter of `$natural:-1`, the database returns documents in reverse natural order.

ObjectId A special 12-byte *BSON* type that has a high probability of being unique when generated. The most significant digits in an ObjectId represent the time when the Object. MongoDB uses ObjectId values as the default values for *_id* fields.

operator A keyword beginning with a `$` used to express a complex query, update, or data transformation. For example, `$gt` is the query language's "greater than" operator. See the `http://docs.mongodb.org/manual/reference/operators` for more information about the available operators.

See Also:

`http://docs.mongodb.org/manual/reference/operators`.

oplog A *capped collection* that stores an ordered history of logical writes to a MongoDB database. The oplog is the basic mechanism enabling *replication* in MongoDB.

See Also:

Oplog Sizes (page 36) and *Change the Size of the Oplog* (page 85).

padding The extra space allocated to document on the disk to prevent moving a document when it grows as the result of `update()` operations.

padding factor An automatically-calibrated constant used to determine how much extra space MongoDB should allocate per document container on disk. A padding factor of 1 means that MongoDB will allocate only the amount of space needed for the document. A padding factor of 2 means that MongoDB will allocate twice the amount of space required by the document.

page fault The event that occurs when a process requests stored data (i.e. a page) from memory that the operating system has moved to disk.

See Also:

Storage FAQ: What are page faults? (page 438)

partition A distributed system architecture that splits data into ranges. *Sharding* is a kind of partitioning.

pcap A packet capture format used by `mongosniff` to record packets captured from network interfaces and display them as human-readable MongoDB operations.

PID A process identifier. On UNIX-like systems, a unique integer PID is assigned to each running process. You can use a PID to inspect a running process and send signals to it.

pipe A communication channel in UNIX-like systems allowing independent processes to send and receive data. In the UNIX shell, piped operations allow users to direct the output of one command into the input of another.

pipeline The series of operations in the *aggregation* process.

See Also:

Aggregation Framework (page 253).

polygon MongoDB's *geospatial* indexes and querying system allow you to build queries around multi-sided polygons on two-dimensional coordinate systems. These queries use the `$within` operator and a sequence of points that define the corners of the polygon.

powerOf2Sizes A per-*collection* setting that changes and normalizes the way that MongoDB allocates space for each *document* in an effort to maximize storage reuse reduce fragmentation. This is the default for *TTL Collections* (page 296). See `collMod` and `usePowerOf2Sizes` for more information. New in version 2.2.

pre-splitting An operation, performed before inserting data that divides the range of possible shard key values into chunks to facilitate easy insertion and high write throughput. When deploying a *sharded cluster*, in some cases pre-splitting will expedite the initial distribution of documents among shards by manually dividing the collection into chunks rather than waiting for the MongoDB *balancer* to create chunks during the course of normal operation.

primary In a *replica set*, the primary member is the current *master* instance, which receives all write operations.

primary key A record's unique, immutable identifier. In an *RDBMS*, the primary key is typically an integer stored in each row's `id` field. In MongoDB, the *_id* field holds a document's primary key which is usually a BSON *ObjectId*.

primary shard For a database where *sharding* is enabled, the primary shard holds all un-sharded collections.

priority In the context of *replica sets*, priority is a configurable value that helps determine which nodes in a replica set are most likely to become *primary*.

See Also:

Replica Set Node Priority (page 34)

projection A document given to a *query* that specifies which fields MongoDB will return from the documents in the result set.

query A read request. MongoDB queries use a *JSON*-like query language that includes a variety of *query operators* with names that begin with a `$` character. In the `mongo` shell, you can issue queries using the `db.collection.find()` and `db.collection.findOne()` methods.

query optimizer For each query, the MongoDB query optimizer generates a query plan that matches the query to the index that produces the fastest results. The optimizer then uses the query plan each time the `mongod` receives the query. If a collection changes significantly, the optimizer creates a new query plan.

RDBMS Relational Database Management System. A database management system based on the relational model, typically using *SQL* as the query language.

read preference A setting on the MongoDB *drivers* (page 285) that determines how the clients direct read operations. Read preference affects all replica sets including shards. By default, drivers direct all reads to *primary* nodes for *strict consistency*. However, you may also direct reads to secondary nodes for *eventually consistent* reads.

See Also:

Read Preference (page 55)

read-lock In the context of a reader-writer lock, a lock that while held allows concurrent readers, but no writers.

record size The space allocated for a document including the padding.

recovering A *replica set* member status indicating that a member is synchronizing or re-synchronizing its data from the primary node. Recovering nodes are unavailable for reads.

replica pairs The precursor to the MongoDB *replica sets*. Deprecated since version 1.6.

replica set A cluster of MongoDB servers that implements master-slave replication and automated failover. MongoDB's recommended replication strategy.

See Also:

Replication (page 31) and *Replication Fundamentals* (page 33).

replication A feature allowing multiple database servers to share the same data, thereby ensuring redundancy and facilitating load balancing. MongoDB supports two flavors of replication: master-slave replication and replica sets.

See Also:

replica set, *sharding*, *Replication* (page 31). and *Replication Fundamentals* (page 33).

replication lag The length of time between the last operation in the primary's *oplog* last operation applied to a particular *secondary* or *slave* node. In general, you want to keep replication lag as small as possible.

See Also:

Replication Lag (page 47)

resident memory The subset of an application's memory currently stored in physical RAM. Resident memory is a subset of *virtual memory*, which includes memory mapped to physical RAM and to disk.

REST An API design pattern centered around the idea of resources and the *CRUD* operations that apply to them. Typically implemented over HTTP. MongoDB provides a simple HTTP REST interface that allows HTTP clients to run commands against the server.

rollback A process that, in certain replica set situations, reverts writes operations to ensure the consistency of all replica set members.

secondary In a *replica set*, the `secondary` members are the current *slave* instances that replicate the contents of the master database. Secondary members may handle read requests, but only the *primary* members can handle write operations.

secondary index A database *index* that improves query performance by minimizing the amount of work that the query engine must perform to fulfill a query.

set name In the context of a *replica set*, the `set name` refers to an arbitrary name given to a replica set when it's first configured. All members of a replica set must have the same name specified with the `replSet` (page 628) setting (or `--replSet` (page 586) option for `mongod`.)

See Also:

replication, *Replication* (page 31) and *Replication Fundamentals* (page 33).

shard A single replica set that stores some portion of a sharded cluster's total data set. See *sharding*.

See Also:

The documents in the *Sharding* (page 105) section of manual.

shard key In a sharded collection, a shard key is the field that MongoDB uses to distribute documents among members of the *sharded cluster*.

sharded cluster The set of nodes comprising a *sharded* MongoDB deployment. A sharded cluster consists of three config processes, one or more replica sets, and one or more `mongos` (page 676) routing processes.

See Also:

The documents in the *Sharding* (page 105) section of manual.

sharding A database architecture that enable horizontal scaling by splitting data into key ranges among two or more replica sets. This architecture is also known as "range-based partitioning." See *shard*.

See Also:

The documents in the *Sharding* (page 105) section of manual.

shell helper A number of `database commands` have "helper" methods in the `mongo` shell that provide a more concise syntax and improve the general interactive experience.

See Also:

mongo (page 591) and `http://docs.mongodb.org/manual/reference/javascript`.

single-master replication A *replication* topology where only a single database instance accepts writes. Single-master replication ensures consistency and is the replication topology employed by MongoDB.

slave In conventional *master*/slave replication, slaves are read-only instances that replicate operations from the *master* database. Data read from slave instances may not be completely consistent with the master. Therefore, applications requiring consistent reads must read from the master database instance.

split The division between *chunks* in a *sharded cluster*.

SQL Structured Query Language (SQL) is a common special-purpose programming language used for interaction with a relational database including access control as well as inserting, updating, querying, and deleting data. There are some similar elements in the basic SQL syntax supported by different database vendors, but most implementations have their own dialects, data types, and interpretations of proposed SQL standards. Complex SQL is generally not directly portable between major *RDBMS* products. Often, `SQL` is often used as a metonym for relational databases.

SSD Solid State Disk. A high-performance disk drive that uses solid state electronics for persistence, as opposed to the rotating platters and movable read/write heads used by traditional mechanical hard drives.

standalone In MongoDB, a standalone is an instance of `mongod` that is running as a single server and not as part of a *replica set*.

strict consistency A property of a distributed system requiring that all nodes always reflect the latest changes to the system. In a database system, this means that any system that can provide data must reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondary nodes have *eventual consistency*.

syslog On UNIX-like systems, a logging process that provides a uniform standard for servers and processes to submit logging information.

tag One or more labels applied to a given replica set member that clients may use to issue data-center aware operations.

TSV A text-based data format consisting of tab-separated values. This format is commonly used to exchange database between relational databases, since the format is well-suited to tabular data. You can import TSV files using `mongoimport`.

TTL Stands for "time to live," and represents an expiration time or period for a given piece of information to remain in a cache or other temporary storage system before the system deletes it or ages it out.

unique index An index that enforces uniqueness for a particular field across a single collection.

upsert A kind of update that either updates the first document matched in the provided query selector or, if no document matches, inserts a new document having the fields implied by the query selector and the update operation.

virtual memory An application's working memory, typically residing on both disk an in physical RAM.

working set The collection of data that MongoDB uses regularly. This data is typically (or preferably) held in RAM.

write concern Specifies whether a write operation has succeeded. Write concern allows your application to detect insertion errors or unavailable `mongod` instances. For *replica sets*, you can configure write concern to confirm replication to a specified number of members.

See Also:

Write Concern for Replica Sets (page 54).

write-lock A lock on the database for a given writer. When a process writes to the database, it takes an exclusive write-lock to prevent other processes from writing or reading.

writeBacks The process within the sharding system that ensures that writes issued to a *shard* that isn't responsible for the relevant chunk, get applied to the proper shard.

See Also:

The *genindex* may provide useful insight into the reference material in this manual.

CHAPTER

FORTYONE

RELEASE NOTES

Always install the latest, stable version of MongoDB. See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

Current stable release (v2.2-series):

41.1 Release Notes for MongoDB 2.2

See the `full index of this page` for a complete list of changes included in 2.2.

41.1.1 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.

Always upgrade to the latest point release in the 2.2 point release. Currently the latest release of MongoDB is 2.2.1.

Synopsis

- `mongod`, 2.2 is a drop-in replacement for 2.0 and 1.8.
- Check your *driver* (page 285) 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 126) section for more information.
- upgrade all `mongos` (page 676) 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` (page 676) 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.
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 34) 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 126).
- Upgrade all `mongos` (page 676) instances *first*, in any order.
- Upgrade all of the `mongod` config server instances using the *stand alone* (page 694) procedure. To keep the cluster online, be 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 694) 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-6902 for more information.

41.1.2 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 `db.collection.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 Framework* (page 253)
- Reference: *Aggregation Framework Reference* (page 269)
- Examples: *Aggregation Framework Examples* (page 259)

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 296) tutorial.

Concurrency Improvements

MongoDB 2.2 increases the server's capacity for concurrent operations with the following improvements:

1. DB Level Locking
2. Improved Yielding on Page Faults
3. Improved Page Fault Detection on Windows

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use, see *locks* (page 638) and *recordStats* (page 650) in *server status* (page 637) and see *current operation output* (page 668), `db.currentOp()`, *mongotop* (page 615), and *mongostat* (page 611).

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.

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* (page 55) and *write concern* (page 54). 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 the wiki page for tag aware sharding.

Fully Supported Read Preference Semantics

All MongoDB clients and drivers now support full *read preferences* (page 55), including consistent support for a full range of *read preference modes* (page 56) and *tag sets* (page 58). This support extends to the `mongos` (page 676) 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` (page 676) 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 694).

`findAndModify` Returns Null Value for Upserts

In version 2.2, for *upsert* operations, `findAndModify` commands will now return the following output:

```
{'ok': 1.0, 'value': null}
```

In the `mongo` shell, `findAndModify` operations running as upserts will only output a `null` value.

Previously, in version 2.0 these operations would return an empty document, e.g. `{ }`.

See: SERVER-6226 for more information.

`mongodump` Output can only Restore to 2.2 MongoDB Instances

If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you may only restore that dump to a 2.2 database.

See: SERVER-6961 for more information.

`ObjectId().toString()` Returns String Literal `ObjectId("...")`

In version 2.2, the `ObjectId.toString()` (page 527) method returns the string representation of the *ObjectId()* (page 288) object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` (page 527) 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 288), which holds the hexadecimal string value in both versions.

`ObjectId().valueOf()` Returns hexadecimal string

In version 2.2, the `ObjectId.valueOf()` (page 528) method returns the value of the *ObjectId()* (page 288) object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` (page 528) 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 288) 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 (e.g. `""`).

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-4442 and the *Are there any restrictions on the names of Collections?* (page 421) FAQ item.

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-4584 and SERVER-6729 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-5516 for more information.

New `$elemMatch` Projection Operator

The `$elemMatch` (page 469) operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the *$elemMatch (projection)* (page 469) documentation and the SERVER-2238 and SERVER-828 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-5648 for more information.

Service Support for `mongos.exe`

You may now run `mongos.exe` instances as a Windows Service. See the *mongos.exe* (page 595) reference and *MongoDB as a Windows Service* (page 24) and SERVER-1589 for more information.

Log Rotate Command Support

MongoDB for Windows now supports log rotation by way of the `logRotate` database command. See SERVER-2612 for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency

Labeled "2008+" on the Downloads Page, 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-3844 for more information.

Tool Improvements

Index Definitions Handled by `mongodump` and `mongorestore`

When you specify the *--collection* (page 598) 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-808 for more information.

`mongorestore` now includes the *--noIndexRestore* (page 601) option to provide the preceding behavior. Use *--noIndexRestore* (page 601) to prevent `mongorestore` from building previous indexes.

`mongooplog` for Replaying Oplogs

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-3873 case and the *mongooplog* (page 603) documentation.

Authentication Support for `mongotop` and `mongostat`

`mongotop` and `mongostat` now contain support for username/password authentication. See SERVER-3875 and SERVER-3871 for more information regarding this change. Also consider the documentation of the following options for additional information:

- *mongotop --username* (page 616)
- *mongotop --password* (page 616)
- *mongostat --username* (page 612)
- *mongostat --password* (page 612)

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* (page 608) option will produce an error rather than silently continue importing data. See SERVER-3937 for more information.

In `mongorestore`, the *--w* (page 601) 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-3854 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 ins size. See SERVER-4593 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:

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483 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-4312 for more information.
- Multi-line command support in shell history. See SERVER-3470 for more information.
- Windows support for the `edit` command. See SERVER-3998 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-1651 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-3819 and SERVER-2395 for more information.

Operations

Support for Logging to Syslog

See the SERVER-2957 case and the documentation of the `syslog` (page 623) run-time option or the *`mongod --syslog`* (page 582) and *`mongos --syslog`* (page 589) command line-options.

`touch` Command

Added the `touch` command to read the data and/or indexes from a collection into memory. See: SERVER-2023 and `touch` for more information.

`indexCounters` No Longer Report Sampled Data

`indexCounters` (page 643) now report actual counters that reflect index use and state. In previous versions, these data were sampled. See SERVER-5784 and `indexCounters` (page 643) for more information.

Padding Specifiable on `compact` Command

See the documentation of the `compact` and the SERVER-4018 issue for more information.

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-3829 and SERVER-5172 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-188 and SERVER-4683. For more information about TCMalloc, see the documentation of TCMalloc 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-3575 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 `members[n].buildIndexes` (page 662) set to `true`, other members of the replica set will *not* sync from this member, unless they also have `members[n].buildIndexes` (page 662) set to `true`. See SERVER-4160 for more information.

New Option To Configure Index Pre-Fetching during Replication

By default, when replicating options, *secondaries* will pre-fetch *Indexes* (page 229) associated with a query to improve replication throughput in most cases. The `replIndexPrefetch` (page 628) setting and `--replIndexPrefetch` (page 587) option allow administrators to disable this feature or allow the `mongod` to pre-fetch only the index on the `_id` field. See SERVER-6718 for more information.

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce, and
- MapReduce will retry jobs following a config error.

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 136)" documentation and SERVER-1506 for more information.

Migration Thresholds Modified

The *migration thresholds* (page 137) 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 137) documentation for more information.

41.1.3 Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice and the SERVER-4683 for more information.

41.1.4 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 2.2
- All Backwards Incompatible Changes
- All Third Party License Notices

What's New in MongoDB 2.2 Online Conference

- Introduction and Welcome
- The Aggregation Framework
- Concurrency
- Data Center Awareness
- TTL Collections
- Closing Remarks and Q&A

See `http://docs.mongodb.org/manual/release-notes/2.2-changes` for an overview of all changes in 2.2.

Previous stable releases:

41.2 Release Notes for MongoDB 2.0

See the `full index of this page` for a complete list of changes included in 2.0.

41.2.1 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. For more information on 2.0 indexes and on rollback, see Index Versions.

`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-1097.

Journaling is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the *`--nojournal`* (page 584) run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` with journaling, you will see a delay the `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 Page.

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 Page.
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 54) 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 34) procedure.

   `rs.stepDown()` and `replSetStepDown` (page 519) 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.
2. Upgrade `mongos` (page 676) routers in any order.

41.2.2 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-2563 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 structures. 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.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0.

Sharding Authentication

Applications can now use authentication with *sharded clusters*.

Replica Sets

Hidden Nodes in Sharded Clusters

In 2.0, `mongos` (page 676) instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, `mongos` (page 676) if you reconfigured a member as hidden, you *had* to restart `mongos` (page 676) 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 *Member Priority* (page 34) 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 54) 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.

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 *Drivers* (page 285) 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 54).

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 95).

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 87).

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` (page 626) 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.

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`* (page 584) run-time option exists for specifying your own group commit interval. 100ms is the default (same as in 1.8).
- 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 285), 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.

See OP_INSERT.

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 and *mapReduce* (page 509).

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 *mapReduce* (page 509).

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 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.

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 *mongo* (page 591).

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `auth` (page 624)) *all* database commands require authentication, *except* the following commands.

- `isMaster`
- `authenticate`
- `getnonce`
- `buildInfo`
- `ping`
- `isdbgrid`

41.2.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 2.0
- All Backward Incompatible Changes

41.3 Release Notes for MongoDB 1.8

See the `full index of this page` for a complete list of changes included in 1.8.

41.3.1 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 709).
- 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.

Upgrading a Standalone `mongod`

1. Download the v1.8.x binaries from the MongoDB Download Page.
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 Page.

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"
              },
              {
                      "_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.
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.

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*:
   - If the shard is a *replica set*, follow the directions above for *Upgrading a Replica Set* (page 709).
   - 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.
3. For each `mongos` (page 676):
   (a) Shut down the `mongos` (page 676) process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page.
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.
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 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.

41.3.2 Changes

Journaling

MongoDB now supports write-ahead journaling 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 234) 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 243) enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

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.
- Extent allocation improvements.
- Improved *replica set* connectivity for `mongos` (page 676).
- `getLastError` improvements for *sharding*.

1.7.4

- `mongos` (page 676) routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` output options.
- *Sparse Index* (page 234).

1.7.3

- Initial *covered index* (page 243) 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* (page 611).
- 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`* (page 612) option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for `$elemMatch` (page 469) on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` works on primitives in arrays.

1.7.0

- Sharding performance improvements for heavy insert loads.
- Slave delay support for replica sets.
- `getLastErrorDefaults` 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

- 1.8.1, 1.8.0
- 1.7.6, 1.7.5, 1.7.4, 1.7.3, 1.7.2, 1.7.1, 1.7.0

41.3.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 1.8

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

S

T

U

V

W