MongoDB Documentation

Release 2.4.5

MongoDB Documentation Project

August 05, 2013

Contents

Part I

Install MongoDB

Installation Guides

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

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

1.1.1 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 CentOS, Fedora and Red Hat Enterprise Linux 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

Additional installation tutorials:

- `http://docs.mongodb.org/manualtutorial/install-mongodb-on-debian-or-ubuntu-linux`
- *Install MongoDB on Debian* (page 9)
- *Install MongoDB on Ubuntu* (page 6)
- *Install MongoDB on Linux* (page 11)
- *Install MongoDB on OS X* (page 13)
- *Install MongoDB on Windows* (page 16)

1.1.2 Package Options

The 10gen repository contains two packages:

- `mongo-10gen-server`

This package contains the mongod (page 1049) and mongos (page 1061) daemons from the latest **stable** release and associated configuration and init scripts. Additionally, you can use this package to *install daemons from a previous release* (page 4) of MongoDB.

- `mongo-10gen`

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

1.1.3 Install MongoDB

Configure Package Management System (YUM)

Create a `/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 `/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 `/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
```

Install Packages

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

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

When this command completes, you have successfully installed MongoDB!

Manage Installed Versions

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

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

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

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

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

1.1.4 Configure MongoDB

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

This MongoDB instance will store its data files in the `http://docs.mongodb.org/manualvar/lib/mongo` and its log files in `http://docs.mongodb.org/manualvar/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/manualvar/lib/mongo` and `http://docs.mongodb.org/manualvar/log/mongo` directories.

1.1.5 Control MongoDB

Warning: With the introduction of `systemd` in Fedora 15, the control scripts included in the packages available in the 10gen repository are not compatible with Fedora systems. A correction is forthcoming, see SERVER-7285 for more information, and in the mean time use your own control scripts *or* install using the procedure outlined in *Install MongoDB on Linux* (page 11).

Start MongoDB

Start the `mongod` (page 1049) process by issuing the following command (as root, or with `sudo`):

```
service mongod start
```

You can verify that the `mongod` (page 1049) process has started successfully by checking the contents of the log file at `http://docs.mongodb.org/manualvar/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` (page 1049) process by issuing the following command (as root, or with `sudo`):

```
service mongod stop
```

Restart MongoDB

You can restart the `mongod` (page 1049) 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/manualvar/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 1061). `mongos` (page 1061) is only used in sharding deployments and typically do not run on the same systems where `mongod` (page 1049) runs. You can use the `mongodb` script referenced above to derive your own `mongos` (page 1061) control script.

SELinux Considerations

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

- enable access to the relevant ports (e.g. 27017) for SELinux. See *Interfaces and Port Numbers* (page 208) for more information on MongoDB's default ports.
- disable SELinux entirely. This requires a system reboot and may have larger implications for your deployment.

1.1.6 Using MongoDB

Among the tools included in the `mongo-10gen` package, is the `mongo` (page 1066) 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` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)"

1.2 Install MongoDB on Ubuntu

1.2.1 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. Although 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 9) tutorial.

See

Additional installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 3)
- *Install MongoDB on Debian* (page 9)

- *Install MongoDB on Linux* (page 11)
- *Install MongoDB on OS X* (page 13)
- *Install MongoDB on Windows* (page 16)

1.2.2 Package Options

The 10gen repository provides the `mongodb-10gen` package, which contains the latest **stable** release. Additionally you can *install previous releases* (page 7) of MongoDB.

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

1.2.3 Install MongoDB

Configure Package Management System (APT)

The Ubuntu package management tool (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys. Issue the following command to import the 10gen public GPG Key:

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

Create a `/etc/apt/sources.list.d/10gen.list` file using the following command.

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

Now issue the following command to reload your repository:

```
sudo apt-get update
```

Install Packages

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

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

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

Manage Installed Versions

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

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

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

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

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

1.2.4 Configure MongoDB

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

This MongoDB instance will store its data files in the `http://docs.mongodb.org/manualvar/lib/mongodb` and its log files in `http://docs.mongodb.org/manualvar/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/manualvar/lib/mongodb` and `http://docs.mongodb.org/manualvar/log/mongodb` directories.

1.2.5 Controlling MongoDB

Starting MongoDB

You can start the `mongod` (page 1049) process by issuing the following command:

```
sudo service mongodb start
```

You can verify that `mongod` (page 1049) has started successfully by checking the contents of the log file at `http://docs.mongodb.org/manualvar/log/mongodb/mongodb.log`.

Stopping MongoDB

As needed, you may stop the `mongod` (page 1049) process by issuing the following command:

```
sudo service mongodb stop
```

Restarting MongoDB

You may restart the `mongod` (page 1049) 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 1061). `mongos` (page 1061) is only used in sharding deployments and typically do not run on the same systems where `mongod` (page 1049) runs. You can use the `mongodb` script referenced above to derive your own `mongos` (page 1061) control script.

1.2.6 Using MongoDB

Among the tools included with the MongoDB package, is the `mongo` (page 1066) 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` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)"

1.3 Install MongoDB on Debian

1.3.1 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: This tutorial applies to both Debian systems and versions of Ubuntu Linux prior to 9.10 "Karmic" which do not use Upstart. Other Ubuntu users will want to follow the *Install MongoDB on Ubuntu* (page 6) tutorial.

See

Additional installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 3)
- *Install MongoDB on Ubuntu* (page 6)
- *Install MongoDB on Linux* (page 11)
- *Install MongoDB on OS X* (page 13)
- *Install MongoDB on Windows* (page 16)

1.3.2 Package Options

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

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

1.3.3 Install 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 `/etc/apt/sources.list.d/10gen.list` file using the following command.

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

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!

Manage Installed Versions

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

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

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

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

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

1.3.4 Configure MongoDB

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

This MongoDB instance will store its data files in the `http://docs.mongodb.org/manualvar/lib/mongodb` and its log files in `http://docs.mongodb.org/manualvar/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/manualvar/lib/mongodb` and `http://docs.mongodb.org/manualvar/log/mongodb` directories.

1.3.5 Controlling MongoDB

Starting MongoDB

Issue the following command to start `mongod` (page 1049):

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

You can verify that `mongod` (page 1049) has started successfully by checking the contents of the log file at `http://docs.mongodb.org/manualvar/log/mongodb/mongodb.log`.

Stopping MongoDB

Issue the following command to stop `mongod` (page 1049):

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

Restarting MongoDB

Issue the following command to restart `mongod` (page 1049):

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

Controlling `mongos`

As of the current release, there are no *control scripts* for `mongos` (page 1061). `mongos` (page 1061) is only used in sharding deployments and typically do not run on the same systems where `mongod` (page 1049) runs. You can use the `mongodb` script referenced above to derive your own `mongos` (page 1061) control script.

1.3.6 Using MongoDB

Among the tools included with the MongoDB package, is the `mongo` (page 1066) 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` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)"

1.4 Install MongoDB on Linux

1.4.1 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

Additional installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 3)
- *Install MongoDB on Ubuntu* (page 6)
- *Install MongoDB on Debian* (page 9)
- *Install MongoDB on OS X* (page 13)
- *Install MongoDB on Windows* (page 16)

1.4.2 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/manualopt` or `http://docs.mongodb.org/manualusr/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_64-2.4.5.tgz > mongodb.tgz
```

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

```
curl http://downloads.mongodb.org/linux/mongodb-linux-i686-2.4.5.tgz > mongodb.tgz
```

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

```
tar -zxvf mongodb.tgz
```

Optional

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

```
cp -R -n  mongodb-linux-????-??-??/ mongodb
```

You can find the `mongod` (page 1049) binary, and the binaries all of the associated MongoDB utilities, in the `bin/` directory within the extracted directory.

Using MongoDB

Before you start `mongod` (page 1049) for the first time, you will need to create the data directory. By default, `mongod` (page 1049) writes data to the `http://docs.mongodb.org/manualdata/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` (page 1049) process has read and write permissions to this directory. If `mongod` (page 1049) runs under the `mongodb` user account, issue the following command to change the owner of this folder:

```
chown mongodb /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` option to `mongod` (page 1049) and the above command.

The 10gen builds of MongoDB contain no *control scripts* or method to control the `mongod` (page 1049) 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/manualusr/local/bin` or `http://docs.mongodb.org/manualusr/bin` directory for easier use.

For testing purposes, you can start a `mongod` (page 1049) directly in the terminal without creating a control script:

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

Note: The above command assumes that the `mongod` (page 1049) binary is accessible via your system's search path, and that you have created a default configuration file located at `/etc/mongod.conf`.

You must `mongod` (page 1049) with a user account that has read and write permissions to the `dbpath` (page 1118).

Among the tools included with this MongoDB distribution, is the `mongo` (page 1066) 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` (page 1066) 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` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)"

1.5 Install MongoDB on OS X

Platform Support

MongoDB only supports OS X versions 10.6 (Snow Leopard) and later.

Changed in version 2.4.

1.5.1 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` (page 1049)") and associated tools: first using the community package management tools, and second using builds of MongoDB provided by 10gen.

See

Additional installation tutorials:

- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 3)
- *Install MongoDB on Ubuntu* (page 6)
- *Install MongoDB on Debian* (page 9)
- *Install MongoDB on Linux* (page 11)
- *Install MongoDB on Windows* (page 16)

1.5.2 Install 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` (page 1049) process in a terminal (for testing or development) or using a process management tool.

```
mongod
```

Then open the mongo (page 1066) 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 (page 1066) 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 (page 1066)" and "*mongo Shell Methods* (page 944)"

1.5.3 Install 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_64-2.4.5.tgz > mongodb.tgz
```

Note: The mongod (page 1049) 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 mongodb.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 2.4 or the version of MongoDB that you are installing.

You can find the mongod (page 1049) 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 (page 1049) for the first time, you will need to create the data directory. By default, mongod (page 1049) writes data to the `http://docs.mongodb.org/manualdata/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* option to `mongod` (page 1049).

The 10gen builds of MongoDB contain no *control scripts* or method to control the `mongod` (page 1049) 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/manualusr/local/bin` directory for easier use.

For testing purposes, you can start a `mongod` (page 1049) directly in the terminal without creating a control script:

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

Note: This command assumes that the `mongod` (page 1049) binary is accessible via your system's search path, and that you have created a default configuration file located at `/etc/mongod.conf`.

Among the tools included with this MongoDB distribution, is the `mongo` (page 1066) 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` (page 1066):

```
./bin/mongo
```

Note: The `./bin/mongo` command assumes that the `mongo` (page 1066) 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` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)"

1.6 Install MongoDB on Windows

1.6.1 Synopsis

This tutorial provides a method for installing and running the MongoDB server (i.e. "`mongod.exe` (page 1072)") 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.

1.6.2 Procedure

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

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

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 1118) setting for `mongod.exe` (page 1072), 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 Practices and Management* (page 207) page.

Warning: Do not allow `mongod.exe` (page 1072) to be accessible to public networks without running in "Secure Mode" (i.e. `auth` (page 1118).) 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` (page 1066) 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` (page 1066) shell will connect to `mongod.exe` (page 1072) running on the localhost interface and port `27017` by default. At the `mongo.exe` (page 1066) 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` (page 1066)" and "*mongo Shell Methods* (page 944)." If you want to develop applications using .NET, see the documentation of C# and MongoDB for more information.

1.6.3 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` (page 1072) added support for running as a Windows service in version 2.0, and `mongos.exe` (page 1073) 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 1116)) and a *configuration file* (page 1115).

1. Create a specific directory for MongoDB log files:

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

2. Create a configuration file for the `logpath` (page 1116) 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 1117) option. If you do not, `mongod.exe` (page 1072) will delete the contents of the existing log file when starting.

Changed in version 2.2: The default `logpath` (page 1116) and `logappend` (page 1117) behavior changed 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`* option to succeed, you *must* specify a `logpath` (page 1116) setting or the *`--logpath`* 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 1118) specify it in the config file (e.g. `C:\mongodb\mongod.cfg`) on that you specified in the *`--install`* operation. You may also specify *`--dbpath`* on the command line; however, always prefer the configuration file.

If the `dbpath` (page 1118) directory does not exist, `mongod.exe` (page 1072) will not be able to start. The default value for `dbpath` (page 1118) is `\data\db`.

Stop or Remove the MongoDB Service

- To stop the MongoDB service:

```
net stop MongoDB
```

- To remove the MongoDB service:

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

1.7 Install MongoDB Enterprise

New in version 2.2.

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

1.7.1 Required Packages

Changed in version 2.4.4: MongoDB Enterprise uses Cyrus SASL instead of GNU SASL. Earlier 2.4 Enterprise versions use GNU SASL (`libgsasl`) instead. For required packages for the earlier 2.4 versions, see *Earlier 2.4 Versions* (page 20).

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

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

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

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

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

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

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

Earlier 2.4 Versions

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

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

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

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

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

  sudo yum update -y
  ```

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

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

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

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

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

1.7.2 Install MongoDB Enterprise Binaries

When you have installed the required packages, and downloaded the Enterprise packages you can install the packages using the same procedure as a standard *installation of MongoDB on Linux Systems* (page 11).

Note: 10gen also provides `.deb` and `.rpm` packages for Enterprise releases for some platforms that you can use to install MongoDB directly using the `dpkg` and `rpm` utilities.

Download and Extract Package

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

Ubuntu 12.04

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-ubuntu1204-2.4.5.tgz > mongod
tar -zxvf mongodb.tgz
cp -R -n mongodb-linux-x86_64-subscription-ubuntu1204-2.4.5/ mongodb
```

Red Hat Enterprise Linux 6.x

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-rhel62-2.4.5.tgz > mongodb.tg
tar -zxvf mongodb.tgz
cp -R -n mongodb-linux-x86_64-subscription-rhel62-2.4.5/ mongodb
```

Amazon Linux AMI

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-amzn64-2.4.5.tgz > mongodb.tg
tar -zxvf mongodb.tgz
cp -R -n mongodb-linux-x86_64-subscription-amzn64-2.4.5/ mongodb
```

SUSE Enterprise Linux

```
curl http://downloads.10gen.com/linux/mongodb-linux-x86_64-subscription-suse11-2.4.5.tgz > mongodb.tg
tar -zxvf mongodb.tgz
cp -R -n mongodb-linux-x86_64-subscription-suse11-2.4.5/ mongodb
```

1.7.3 Running and Using MongoDB

Before you start mongod (page 1049) for the first time, you will need to create the data directory. By default, mongod (page 1049) writes data to the http://docs.mongodb.org/manualdata/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 (page 1049) process has read and write permissions to this directory. If mongod (page 1049) runs under the mongodb user account, issue the following command to change the owner of this folder:

```
chown mongodb /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* option to mongod (page 1049) and the above command.

The 10gen builds of MongoDB contain no *control scripts* or method to control the mongod (page 1049) 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/manualusr/local/bin or http://docs.mongodb.org/manualusr/bin directory for easier use.

For testing purposes, you can start a mongod (page 1049) directly in the terminal without creating a control script:

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

Note: The above command assumes that the mongod (page 1049) binary is accessible via your system's search path, and that you have created a default configuration file located at /etc/mongod.conf.

You must mongod (page 1049) with a user account that has read and write permissions to the dbpath (page 1118).

Among the tools included with this MongoDB distribution, is the mongo (page 1066) 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 (page 1066) 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 (page 1066) 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 (page 1066)" and "*mongo Shell Methods* (page 944)"

1.7.4 Further Reading

As you begin to use MongoDB, consider the *Getting Started with MongoDB* (page 29) and *MongoDB Tutorials* (page 171) resources. To read about features only available in MongoDB Enterprise, consider: *Monitor MongoDB with SNMP* (page 185) and *Deploy MongoDB with Kerberos Authentication* (page 228).

CHAPTER 2

Upgrade MongoDB

2.1 Upgrade to the Latest Revision of MongoDB

Revisions provide security patches, bug fixes, and new or changed features that do not contain any backward breaking changes. Always upgrade to the latest revision in your release series. The third number in the *MongoDB version number* (page 1225) indicates the revision.

2.1.1 Before Upgrading

- Ensure you have an up-to-date backup of your data set. See *Backup Strategies for MongoDB Systems* (page 133).
- Consult the following documents for any special considerations or compatibility issues specific to your MongoDB release:
  - The release notes, located at *Release Notes* (page 1165).
  - The documentation for your driver. See *MongoDB Drivers and Client Libraries* (page 575).
- If your installation includes *replica sets*, plan the upgrade during a predefined maintenance window.
- Before you upgrade a production environment, use the procedures in this document to upgrade a *staging* environment that reproduces your production environment, to ensure that your production configuration is compatible with all changes.

2.1.2 Upgrade Procedure

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

Upgrade each `mongod` (page 1049) and `mongos` (page 1061) binary separately, using the procedure described here. When upgrading a binary, use the procedure *Upgrade a MongoDB Instance* (page 24).

Follow this upgrade procedure:

1. For deployments that use authentication, first upgrade all of your MongoDB *drivers* (page 575). To upgrade, see the documentation for your driver.
2. Upgrade sharded clusters, as described in *Upgrade Sharded Clusters* (page 24).
3. Upgrade any standalone instances. See *Upgrade a MongoDB Instance* (page 24).
4. Upgrade any replica sets that are not part of a sharded cluster, as described in *Upgrade Replica Sets* (page 25).

2.1.3 Upgrade a MongoDB Instance

To upgrade a `mongod` (page 1049) or `mongos` (page 1061) instance, use one of the following approaches:

- Upgrade the instance using the operating system's package management tool and the packages provided by 10gen. This is the preferred approach. See *Install MongoDB* (page 3).
- Upgrade the instance by replacing the existing binaries with new binaries. See *Replace the Existing Binaries* (page 24).

2.1.4 Replace the Existing Binaries

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

This section describes how to upgrade MongoDB by replacing the existing binaries. The preferred approach to an upgrade is to use the operating system's package management tool and the packages provided by 10gen, as described in *Install MongoDB* (page 3).

To upgrade a `mongod` (page 1049) or `mongos` (page 1061) instance by replacing the existing binaries:

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

2.1.5 Upgrade Sharded Clusters

To upgrade a sharded cluster:

1. Disable the cluster's balancer, as described in *Disable the Balancer* (page 543).
2. Upgrade each `mongos` (page 1061) instance by following the instructions below in *Upgrade a MongoDB Instance* (page 24). You can upgrade the `mongos` (page 1061) instances in any order.
3. Upgrade each `mongod` (page 1049) *config server* (page 498) individually starting with the last config server listed in your *mongos --configdb* string and working backward. To keep the cluster online, make sure at least one config server is always running. For each config server upgrade, follow the instructions below in *Upgrade a MongoDB Instance* (page 24)

   ..example:: Given the following config string:

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

   You would upgrade the config servers in the following order:

   (a) cfg2.example.net

   (b) cfg1.example.net

   (c) cfg0.example.net

4. Upgrade each shard.
   - If a shard is a replica set, upgrade the shard using the procedure below titled *Upgrade Replica Sets* (page 25).

   - If a shard is a standalone instance, upgrade the shard using the procedure below titled *Upgrade a MongoDB Instance* (page 24).

5. Re-enable the balancer, as described in *Enable the Balancer* (page 544).

2.1.6 Upgrade Replica Sets

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

Upgrade Secondaries

Upgrade each secondary separately as follows:

1. Upgrade the secondary's `mongod` (page 1049) binary by following the instructions below in *Upgrade a MongoDB Instance* (page 24).
2. After upgrading a secondary, wait for the secondary to recover to the `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` (page 1017) in the `mongo` (page 1066) shell.

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

Upgrade the Primary

1. Step down the primary to initiate the normal *failover* (page 388) procedure. Using one of the following:
   - The `rs.stepDown()` (page 1018) helper in the `mongo` (page 1066) shell.
   - The `replSetStepDown` (page 868) database command.

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

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

2. Once the primary has stepped down, call the `rs.status()` (page 1017) method from the `mongo` (page 1066) shell until you see that another member has assumed the `PRIMARY` state.
3. Shut down the original primary and upgrade its instance by following the instructions below in *Upgrade a MongoDB Instance* (page 24).

To upgrade to a new revision of a MongoDB major release, see *Upgrade to the Latest Revision of MongoDB* (page 23)

CHAPTER 3

Release Notes

You should always install the latest, *stable* version of MongoDB. Stable versions have an even number for the second number in the *version number* (page 1225). The following release notes are for stable versions:

- Current Stable Release:
  - *Release Notes for MongoDB 2.4* (page 1167)
- Previous Stable Releases:
  - *Release Notes for MongoDB 2.2* (page 1189)
  - *Release Notes for MongoDB 2.0* (page 1199)
  - *Release Notes for MongoDB 1.8* (page 1205)

CHAPTER 4

First Steps with MongoDB

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

4.1 Getting Started with MongoDB

This tutorial provides an introduction to basic database operations using the `mongo` (page 1066) shell. `mongo` (page 1066) is a part of the standard MongoDB distribution and provides a full JavaScript environment with a complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB. See the mongo JavaScript API documentation and the `mongo` (page 1066) shell *JavaScript Method Reference* (page 944).

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

This tutorial addresses the following aspects of MongoDB use:

4.1.1 Connect to a Database

In this section, you connect to the database server, which runs as `mongod` (page 1049), and begin using the `mongo` (page 1066) shell to select a logical database within the database instance and access the help text in the `mongo` (page 1066) shell.

Connect to a `mongod`

From a system prompt, start `mongo` (page 1066) by issuing the `mongo` (page 1066) command, as follows:

```
mongo
```

By default, `mongo` (page 1066) looks for a database server listening on port `27017` on the `localhost` interface. To connect to a server on a different port or interface, use the `--port` and `--host` options.

Select a Database

After starting the `mongo` (page 1066) shell your session will use the `test` database by default. At any time, issue the following operation at the `mongo` (page 1066) to report the name of the current database:

```
db
```

1. From the `mongo` (page 1066) shell, display the list of databases, with the following operation:

   ```
   show dbs
   ```

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

   ```
   use mydb
   ```

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

   ```
   db
   ```

   At this point, if you issue the `show dbs` operation again, it will not include the `mydb` database. MongoDB will not permanently create a database until you insert data into that database. The *Create a Collection and Insert Documents* (page 30) section describes the process for inserting data.

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

Display `mongo` Help

At any point, you can access help for the `mongo` (page 1066) shell using the following operation:

```
help
```

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

4.1.2 Create a Collection and Insert Documents

In this section, you insert documents into a new *collection* named `testData` within the new *database* named `mydb`.

MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses *dynamic schemas* (page 732), you also need not specify the structure of your documents before inserting them into the collection.

1. From the `mongo` (page 1066) shell, confirm you are in the `mydb` database by issuing the following:

   ```
   db
   ```

2. If `mongo` (page 1066) does not return `mydb` for the previous operation, set the context to the `mydb` database, with the following operation:

```
use mydb
```

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

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

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

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

When you insert the first document, the `mongod` (page 1049) will create both the `mydb` database and the `testData` collection.

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

```
show collections
```

The `mongo` (page 1066) shell will return the list of the collections in the current (i.e. `mydb`) database. At this point, the only collection is `testData`. All `mongod` (page 1049) databases also have a `system.indexes` (page 1133) collection.

6. Confirm that the documents exist in the `testData` collection by issuing a query on the collection using the `find()` (page 951) method:

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

This operation returns the following results. The *ObjectId* (page 68) values will be unique:

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

All MongoDB documents must have an `_id` field with a unique value. These operations do not explicitly specify a value for the `_id` field, so `mongo` (page 1066) creates a unique *ObjectId* (page 68) value for the field before inserting it into the collection.

4.1.3 Insert Documents using a For Loops or JavaScript Function

To perform the remaining procedures in this tutorial, first add more documents to your database using one or both of the procedures described in *Generate Test Data* (page 34).

4.1.4 Working with the Cursor

When you query a *collection*, MongoDB returns a "cursor" object that contains the results of the query. The `mongo` (page 1066) shell then iterates over the cursor to display the results. Rather than returning all results at once, the shell iterates over the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. In the shell, use enter `it` to iterate over the next set of results.

The procedures in this section show other ways to work with a cursor. For comprehensive documentation on cursors, see *Iterate the Returned Cursor* (page 90).

Iterate over the Cursor with a Loop

Before using this procedure, make sure to add at least 25 documents to a collection using one of the procedures in *Generate Test Data* (page 34). You can name your database and collections anything you choose, but this procedure will assume the database named `test` and a collection named `testData`.

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

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

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

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

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

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

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

Use Array Operations with the Cursor

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

1. In the `mongo` (page 1066) shell, query the `testData` collection and assign the resulting cursor object to the `c` variable:

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

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

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

   MongoDB returns the following:

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

   When you access documents in a cursor using the array index notation, `mongo` (page 1066) first calls the `cursor.toArray()` method and loads into RAM all documents returned by the cursor. The index is then applied to the resulting array. This operation iterates the cursor completely and exhausts the cursor.

   For very large result sets, `mongo` (page 1066) may run out of available memory.

For more information on the cursor, see *Iterate the Returned Cursor* (page 90).

Query for Specific Documents

MongoDB has a rich query system that allows you to select and filter the documents in a collection along specific fields and values. See *Query Document* (page 42) and *Read* (page 83) for a full account of queries in MongoDB.

In this procedure, you query for specific documents in the `testData` *collection* by passing a "query document" as a parameter to the `find()` (page 951) method. A query document specifies the criteria the query must match to return a document.

In the `mongo` (page 1066) shell, query for all documents where the `x` field has a value of `18` by passing the `{ x : 18 }` query document as a parameter to the `find()` (page 951) method:

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

MongoDB returns one document that fits this criteria:

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

Return a Single Document from a Collection

With the `db.collection.findOne()` (page 955) method you can return a single *document* from a MongoDB collection. The `findOne()` (page 955) method takes the same parameters as `find()` (page 951), but returns a document rather than a cursor.

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

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

For more information on querying for documents, see the *Read* (page 83) and *Read Operations* (page 41) documentation.

Limit the Number of Documents in the Result Set

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

To specify the maximum number of documents in the result set, call the `limit()` (page 985) method on a cursor, as in the following command:

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

MongoDB will return the following result, with different *ObjectId* (page 68) values:

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

4.1.5 Next Steps with MongoDB

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

- *CRUD Operations for MongoDB* (page 75)
- *SQL to MongoDB Mapping Chart* (page 1042)
- *MongoDB Drivers and Client Libraries* (page 575)

4.2 Generate Test Data

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

4.2.1 Insert Multiple Documents Using a For Loop

You can add documents to a new or existing collection by using a JavaScript `for` loop run from the `mongo` (page 1066) shell.

1. From the `mongo` (page 1066) shell, insert new documents into the `testData` collection using the following `for` loop. If the `testData` collection does not exist, MongoDB creates the collection implicitly.

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

2. Use find() to query the collection:

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

The `mongo` (page 1066) shell displays the first 20 documents in the collection. Your *ObjectId* (page 68) values will be different:

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

1. The `find()` (page 951) returns a cursor. To iterate the cursor and return more documents use the `it` operation in the `mongo` (page 1066) shell. The `mongo` (page 1066) shell will exhaust the cursor, and return the following documents:

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

4.2.2 Insert Multiple Documents with a `mongo` Shell Function

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

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

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

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

  print(col.count());

}
```

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

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

Store the function in your *.mongorc.js* (page 1069) file. The `mongo` (page 1066) shell loads the function for you every time you start a session.

Example

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

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

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

- *Getting Started with MongoDB* (page 29)
- *Generate Test Data* (page 34)
- *Create* (page 75)
- *Read* (page 83)
- *Update* (page 93)
- *Delete* (page 101)
- *Data Modeling* (page 105)
- *MongoDB Drivers and Client Libraries* (page 575)

Part II

Core MongoDB Operations (CRUD)

CRUD stands for *create*, *read*, *update*, and *delete*, which are the four core database operations used in database driven application development. The *CRUD Operations for MongoDB* (page 75) section provides introduction to each class of operation along with complete examples of each operation. The documents in the *Read and Write Operations in MongoDB* (page 41) section provide a higher level overview of the behavior and available functionality of these operations.

CHAPTER 5

Read and Write Operations in MongoDB

The *Read Operations* (page 41) and *Write Operations* (page 53) documents provide higher level introductions and description of the behavior and operations of read and write operations for MongoDB deployments. The *BSON Documents* (page 61) provides an overview of *documents* and document-orientation in MongoDB.

5.1 Read Operations

Read operations include all operations that return a cursor in response to application data requests (i.e. *queries*,) and also include a number of *aggregation* (page 245) operations that do not return a cursor but have similar properties as queries. These commands include `aggregate` (page 834), `count` (page 834), and `distinct` (page 835).

This document describes the syntax and structure of the queries applications use to request data from MongoDB and how different factors affect the efficiency of reads.

Note: All of the examples in this document use the `mongo` (page 1066) shell interface. All of these operations are available in an idiomatic interface for each language by way of the *MongoDB Driver* (page 575). See your driver documentation for full API documentation.

5.1.1 Queries in MongoDB

In the `mongo` (page 1066) shell, the `find()` (page 951) and `findOne()` (page 955) methods perform read operations. The `find()` (page 951) method has the following syntax: [1]

```
db.collection.find( <query>, <projection> )
```

- The `db.collection` object specifies the database and collection to query. All queries in MongoDB address a *single* collection.

  You can enter `db` in the `mongo` (page 1066) shell to return the name of the current database. Use the `show collections` operation in the `mongo` (page 1066) shell to list the current collections in the database.

- Queries in MongoDB are *BSON* objects that use a set of *query operators* (page 785) to describe query parameters.

  The `<query>` argument of the `find()` (page 951) method holds this query document. A read operation without a query document will return all documents in the collection.

[1] `db.collection.find()` (page 951) is a wrapper for the more formal query structure with the `$query` operator.

- The `<projection>` argument describes the result set in the form of a document. Projections specify or limit the fields to return.

  Without a projection, the operation will return all fields of the documents. Specify a projection if your documents are larger, or when your application only needs a subset of available fields.

- The order of documents returned by a query is not defined and is not necessarily consistent unless you specify a sort (`sort()` (page 991)).

For example, the following operation on the `inventory` collection selects all documents where the `type` field equals `'food'` and the `price` field has a value less than `9.95`. The projection limits the response to the `item` and `qty`, and `_id` field:

```
db.inventory.find( { type: 'food', price: { $lt: 9.95 } },
                   { item: 1, qty: 1 } )
```

The `findOne()` (page 955) method is similar to the `find()` (page 951) method except the `findOne()` (page 955) method returns a single document from a collection rather than a cursor. The method has the syntax:

```
db.collection.findOne( <query>, <projection> )
```

For additional documentation and examples of the main MongoDB read operators, refer to the *Read* (page 83) page of the *Core MongoDB Operations (CRUD)* (page 39) section.

Query Document

This section provides an overview of the query document for MongoDB queries. See the preceding section for more information on *queries in MongoDB* (page 41).

The following examples demonstrate the key properties of the query document in MongoDB queries, using the `find()` (page 951) method from the `mongo` (page 1066) shell, and a collection of documents named `inventory`:

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

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

  Not specifying a query document to the `find()` (page 951) is equivalent to specifying an empty query document. Therefore the following operation is equivalent to the previous operation:

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

- A single-clause query selects all documents in a collection where a field has a certain value. These are simple "equality" queries.

  In the following example, the query selects all documents in the collection where the `type` field has the value `snacks`:

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

- A single-clause query document can also select all documents in a collection given a condition or set of conditions for one field in the collection's documents. Use the *query operators* (page 785) to specify conditions in a MongoDB query.

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

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

  Note: Although you can express this query using the `$or` (page 790) operator, choose the `$in` (page 787) operator rather than the `$or` (page 790) operator when performing equality checks on the same field.

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

  In the following example, the query document specifies an equality match on a single field, followed by a range of values for a second field using a *comparison operator* (page 785):

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

  This query selects all documents where the `type` field has the value `'food'` **and** the value of the `price` field is less than (`$lt` (page 788)) `9.95`.

- Using the `$or` (page 790) operator, you can specify a compound query that joins each clause with a logical `OR` conjunction so that the query selects the documents in the collection that match at least one condition.

  In the following example, the query document selects all documents in the collection where the field `qty` has a value greater than (`$gt` (page 786)) `100` **or** the value of the `price` field is less than (`$lt` (page 788)) `9.95`:

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

- With additional clauses, you can specify precise conditions for matching documents. In the following example, the compound query document selects all documents in the collection where the value of the `type` field is `'food'` **and** *either* the `qty` has a value greater than (`$gt` (page 786)) `100` *or* the value of the `price` field is less than (`$lt` (page 788)) `9.95`:

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

Subdocuments

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

- Equality matches within subdocuments select documents if the subdocument matches *exactly* the specified subdocument, including the field order.

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

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

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

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

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

Arrays

When the field holds an array, you can query for values in the array, and if the array holds sub-documents, you query for specific fields within the sub-documents using *dot notation*:

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

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

- Equality matches can specify a single element in the array. If the array contains at least *one* element with the specified value, as in the following example: the query matches all documents where the value of the field `tags` is an array that contains, as one of its elements, the element `'fruit'`:

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

  Equality matches can also select documents by values in an array using the array index (i.e. position) of the element in the array, as in the following example: the query uses the *dot notation* to match all documents where the value of the `tags` field is an array whose first element equals `'fruit'`:

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

In the following examples, consider an array that contains subdocuments:

- If you know the array index of the subdocument, you can specify the document using the subdocument's position.

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

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

- If you do not know the index position of the subdocument, concatenate the name of the field that contains the array, with a dot (`.`) and the name of the field in the subdocument.

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

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

- To match by multiple fields in the subdocument, you can use either dot notation or the `$elemMatch` (page 809) operator:

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

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

The following example uses `$elemMatch` (page 809) to query for documents where the value of the `memos` field is an array that has at least one subdocument that contains the field `memo` equal to `'on time'` and the field `by` equal to `'shipping'`:

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

Refer to the *Query, Update and Projection Operators* (page 785) document for the complete list of query operators.

Result Projections

The *projection* specification limits the fields to return for all matching documents. Restricting the fields to return can minimize network transit costs and the costs of deserializing documents in the application layer.

The second argument to the `find()` (page 951) method is a projection, and it takes the form of a *document* with a list of fields for inclusion or exclusion from the result set. You can either specify the fields to include (e.g. `{ field: 1 }`) or specify the fields to exclude (e.g. `{ field: 0 }`). The `_id` field is, by default, included in the result set. To exclude the `_id` field from the result set, you need to specify in the projection document the exclusion of the `_id` field (i.e. `{ _id: 0 }`).

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

Consider the following projection specifications in `find()` (page 951) operations:

- If you specify no projection, the `find()` (page 951) method returns all fields of all documents that match the query.

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

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

- A projection can explicitly include several fields. In the following operation, `find()` (page 951) method returns all documents that match the query as well as `item` and `qty` fields. The results also include the `_id` field:

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

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

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

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

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

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

  This operation returns all documents where the value of the `type` field is `food`, but does not include the `type` field in the output.

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

The `$elemMatch` (page 824) and `$slice` (page 826) projection operators provide more control when projecting only a portion of an array.

5.1.2 Indexes

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process and thereby simplifying the work associated with fulfilling queries within MongoDB. The indexes themselves are a special data structure that MongoDB maintains when inserting or modifying documents, and any given index can: support and optimize specific queries, sort operations, and allow for more efficient storage utilization. For more information about indexes in MongoDB see: *Indexes* (page 307) and *Indexing Overview* (page 309).

You can create indexes using the `db.collection.ensureIndex()` (page 949) method in the `mongo` (page 1066) shell, as in the following prototype operation:

```
db.collection.ensureIndex( { <field1>: <order>, <field2>: <order>, ... } )
```

- The `field` specifies the field to index. The field may be a field from a subdocument, using *dot notation* to specify subdocument fields.

  You can create an index on a single field or a *compound index* (page 311) that includes multiple fields in the index.

- The `order` option is specifies either ascending (`1`) or descending (`-1`).

  MongoDB can read the index in either direction. In most cases, you only need to specify *indexing order* (page 312) to support sort operations in compound queries.

Covering a Query

An index *covers* (page 322) a query, a *covered query*, when:

- all the fields in the *query* (page 42) are part of that index, **and**
- all the fields returned in the documents that match the query are in the same index.

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

Example

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

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

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

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

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

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

See *Create Indexes that Support Covered Queries* (page 322) for more information on the behavior and use of covered queries.

Measuring Index Use

The `explain()` (page 979) cursor method allows you to inspect the operation of the query system, and is useful for analyzing the efficiency of queries, and for determining how the query uses the index. Call the `explain()` (page 979) method on a cursor returned by `find()` (page 951), as in the following example:

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

Note: Only use `explain()` (page 979) to test the query operation, and *not* the timing of query performance. Because `explain()` (page 979) attempts multiple query plans, it does not reflect accurate query performance.

If the above operation could not use an index, the output of `explain()` (page 979) would resemble the following:

```
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 5,
  "nscannedObjects" : 4000006,
  "nscanned" : 4000006,
  "nscannedObjectsAllPlans" : 4000006,
  "nscannedAllPlans" : 4000006,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 2,
  "nChunkSkips" : 0,
  "millis" : 1591,
  "indexBounds" : { },
  "server" : "mongodb0.example.net:27017"
}
```

The `BasicCursor` value in the `cursor` (page 982) field confirms that this query does not use an index. The `explain.nscannedObjects` (page 982) value shows that MongoDB must scan 4,000,006 documents to return only 5 documents. To increase the efficiency of the query, create an index on the `type` field, as in the following example:

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

Run the `explain()` (page 979) operation, as follows, to test the use of the index:

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

Consider the results:

```
{
  "cursor" : "BtreeCursor type_1",
  "isMultiKey" : false,
  "n" : 5,
  "nscannedObjects" : 5,
  "nscanned" : 5,
  "nscannedObjectsAllPlans" : 5,
  "nscannedAllPlans" : 5,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : { "type" : [
                                [ "food",
```

```
                                        "food" ]
                              ] },
  "server" : "mongodbo0.example.net:27017" }
```

The BtreeCursor value of the cursor (page 982) field indicates that the query used an index. This query:

- returned 5 documents, as indicated by the n (page 982) field;
- scanned 5 documents from the index, as indicated by the nscanned (page 982) field;
- then read 5 full documents from the collection, as indicated by the nscannedObjects (page 982) field.

  Although the query uses an index to find the matching documents, if indexOnly (page 983) is false then an index could not *cover* (page 46) the query: MongoDB could not both match the *query conditions* (page 42) **and** return the results using only this index. See *Create Indexes that Support Covered Queries* (page 322) for more information.

Query Optimization

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

To create a new query plan, the query optimizer:

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

   If an index returns a result already returned by another index, the optimizer skips the duplicate match. In the case of the two buffers, both buffers are de-duped.
3. stops the testing of candidate plans and selects an index when one of the following events occur:
   - An *unordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned a threshold number of matching results:
     - Version 2.0: Threshold is the query batch size. The default batch size is 101.
     - Version 2.2: Threshold is 101.

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

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

To manually compare the performance of a query using more than one index, you can use the hint() (page 985) and explain() (page 979) methods in conjunction, as in the following prototype:

```
db.collection.find().hint().explain()
```

The following operations each run the same query but will reflect the use of the different indexes:

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

This returns the statistics regarding the execution of the query. For more information on the output of `explain()` (page 979), see *cursor.explain()* (page 979).

Note: If you run `explain()` (page 979) without including `hint()` (page 985), the query optimizer reevaluates the query and runs against multiple indexes before returning the query statistics.

As collections change over time, the query optimizer deletes a query plan and reevaluates the after any of the following events:

- the collection receives 1,000 write operations.
- the `reIndex` (page 894) rebuilds the index.
- you add or drop an index.
- the `mongod` (page 1049) process restarts.

For more information, see *Indexing Strategies* (page 321).

Query Operations that Cannot Use Indexes Effectively

Some query operations cannot use indexes effectively or cannot use indexes at all. Consider the following situations:

- The inequality operators `$nin` (page 789) and `$ne` (page 789) are not very selective, as they often match a large portion of the index.

  As a result, in most cases, a `$nin` (page 789) or `$ne` (page 789) query with an index may perform no better than a `$nin` (page 789) or `$ne` (page 789) query that must scan all documents in a collection.

- Queries that specify regular expressions, with inline JavaScript regular expressions or `$regex` (page 798) operator expressions, cannot use an index. *However*, the regular expression with anchors to the beginning of a string *can* use an index.

5.1.3 Cursors

The `find()` (page 951) method returns a *cursor* to the results; however, in the `mongo` (page 1066) shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times [2] to print up to the first 20 documents that match the query, as in the following example:

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

When you assign the `find()` (page 951) to a variable:

- you can call the cursor variable in the shell to iterate up to 20 times [2] and print the matching documents, as in the following example:

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

  myCursor
  ```

- you can use the cursor method `next()` (page 989) to access the documents, as in the following example:

[2] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Executing Queries* (page 606) for more information.

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

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

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

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

- you can use the cursor method `forEach()` (page 984) to iterate the cursor and access the documents, as in the following example:

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

myCursor.forEach(printjson);
```

See *JavaScript cursor methods* (page 977) and your *driver* (page 575) documentation for more information on cursor methods.

Iterator Index

In the `mongo` (page 1066) shell, you can use the `toArray()` (page 992) method to iterate the cursor and return the documents in an array, as in the following:

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

The `toArray()` (page 992) method loads into RAM all documents returned by the cursor; the `toArray()` (page 992) method exhausts the cursor.

Additionally, some *drivers* (page 575) provide access to the documents by using an index on the cursor (i.e. `cursor[index]`). This is a shortcut for first calling the `toArray()` (page 992) method and then using an index on the resulting array.

Consider the following example:

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

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

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

Cursor Behaviors

Consider the following behaviors related to cursors:

- By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor. To override this behavior, you can specify the `noTimeout` wire protocol flag in your query; however, you should either close the cursor manually or exhaust the cursor. In the `mongo` (page 1066) shell, you can set the `noTimeout` flag:

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

See your *driver* (page 575) documentation for information on setting the `noTimeout` flag. See *Cursor Flags* (page 51) for a complete list of available cursor flags.

- Because the cursor is not isolated during its lifetime, intervening write operations may result in a cursor that returns a single document [3] more than once. To handle this situation, see the information on *snapshot mode* (page 743).
- The MongoDB server returns the query results in batches:
  - For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `batchSize()` (page 978) and `limit()` (page 985).
  - For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort and will return all documents in the first batch.
  - Batch size will not exceed the *maximum BSON document size* (page 1139).
  - As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` (page 989) will perform a `getmore operation` (page 1000) to retrieve the next batch.

    To see how many documents remain in the batch as you iterate the cursor, you can use the `objsLeftInBatch()` (page 989) method, as in the following example:

    ```
    var myCursor = db.inventory.find();

    var myFirstDocument = myCursor.hasNext() ? myCursor.next() : null;

    myCursor.objsLeftInBatch();
    ```

- You can use the command `cursorInfo` (page 906) to retrieve the following information on cursors:
  - total number of open cursors
  - size of the client cursors in current use
  - number of timed out cursors since the last server restart

  Consider the following example:

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

  The result from the command returns the following document:

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

Cursor Flags

The `mongo` (page 1066) shell provides the following cursor flags:

- `DBQuery.Option.tailable`

[3] A single document relative to value of the `_id` field. A cursor cannot return the same document more than once *if* the document has not changed.

- DBQuery.Option.slaveOk
- DBQuery.Option.oplogReplay
- DBQuery.Option.noTimeout
- DBQuery.Option.awaitData
- DBQuery.Option.exhaust
- DBQuery.Option.partial

Aggregation

Changed in version 2.2.

MongoDB can perform some basic data aggregation operations on results before returning data to the application. These operations are not queries; they use *database commands* rather than queries, and they do not return a cursor. However, they still require MongoDB to read data.

Running aggregation operations on the database side can be more efficient than running them in the application layer and can reduce the amount of data MongoDB needs to send to the application. These aggregation operations include basic grouping, counting, and even processing data using a map reduce framework. Additionally, in 2.2 MongoDB provides a complete aggregation framework for more rich aggregation operations.

The aggregation framework provides users with a "pipeline" like framework: documents enter from a collection and then pass through a series of steps by a sequence of *pipeline operators* (page 264) that manipulate and transform the documents until they're output at the end. The aggregation framework is accessible via the `aggregate` (page 834) command or the `db.collection.aggregate()` (page 945) helper in the `mongo` (page 1066) shell.

For more information on the aggregation framework see *Aggregation* (page 245).

Additionally, MongoDB provides a number of simple data aggregation operations for more basic data aggregation operations:

- `count` (page 834) (`count()` (page 978))
- `distinct` (page 835) (`db.collection.distinct()` (page 948))
- `group` (page 836) (`db.collection.group()` (page 958))
- `mapReduce` (page 840). (Also consider `mapReduce()` (page 963) and *Map-Reduce* (page 291).)

5.1.4 Architecture

Read Operations from Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of `mongod` (page 1049) in a way that is nearly transparent to the application. See the *Sharding* (page 485) section of this manual for additional information about these deployments.

For a sharded cluster, you issue all operations to one of the `mongos` (page 1061) instances associated with the cluster. `mongos` (page 1061) instances route operations to the `mongod` (page 1049) in the cluster and behave like `mongod` (page 1049) instances to the application. Read operations to a sharded collection in a sharded cluster are largely the same as operations to a *replica set* or *standalone* instances. See the section on *Read Operations in Sharded Clusters* (page 506) for more information.

In sharded deployments, the `mongos` (page 1061) instance routes the queries from the clients to the `mongod` (page 1049) instances that hold the data, using the cluster metadata stored in the *config database* (page 498).

For sharded collections, if queries do not include the *shard key* (page 502), the `mongos` (page 1061) must direct the query to all shards in a collection. These *scatter gather* queries can be inefficient, particularly on larger clusters, and are unfeasible for routine operations.

For more information on read operations in sharded clusters, consider the following resources:

- *An Introduction to Shard Keys* (page 502)
- *Shard Key Internals and Operations* (page 502)
- *Querying Sharded Clusters* (page 503)
- *Sharded Cluster Query Routing* (page 506)

Read Operations from Replica Sets

Replica sets use *read preferences* to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can modify that behavior by changing the *read preference mode* (page 399).

You can configure the *read preference mode* (page 399) on a per-connection or per-operation basis to allow reads from *secondaries* to:

- reduce latency in multi-data-center deployments,
- improve read throughput by distributing high read-volumes (relative to write volume),
- for backup operations, and/or
- to allow reads during *failover* (page 388) situations.

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

For more information on read preference or on the read preference modes, see *Read Preference* (page 398) and *Read Preference Modes* (page 399).

5.2 Write Operations

All operations that create or modify data in the MongoDB instance are write operations. MongoDB represents data as *BSON documents* stored in *collections*. Write operations target one collection and are atomic on the level of a single document: no single write operation can atomically affect more than one document or more than one collection.

This document introduces the write operators available in MongoDB as well as presents strategies to increase the efficiency of writes in applications.

5.2.1 Write Operators

For information on write operators and how to write data to a MongoDB database, see the following pages:

- *Create* (page 75)
- *Update* (page 93)
- *Delete* (page 101)

For information on specific methods used to perform write operations in the `mongo` (page 1066) shell, see the following:

- `db.collection.insert()` (page 961)
- `db.collection.update()` (page 974)
- `db.collection.save()` (page 972)
- `db.collection.findAndModify()` (page 952)
- `db.collection.remove()` (page 970)

For information on how to perform write operations from within an application, see the *MongoDB Drivers and Client Libraries* (page 575) documentation or the documentation for your client library.

5.2.2 Write Concern

Write concern is a quality of every write operation issued to a MongoDB deployment, and describes the amount of *concern* the application has for the outcome of the write operation. With weak or disabled write concern, the application can send a write operation to MongoDB and then continue without waiting for a response from the database. With stronger write concerns, write operations wait until MongoDB acknowledges or confirms a successful write operation. MongoDB provides different levels of write concern to better address the specific needs of applications.

Note: The *driver write concern* (page 1223) change created a new connection class in all of the MongoDB drivers, called `MongoClient` with a different default write concern. See the *release notes* (page 1223) for this change, and the release notes for the driver you're using for more information about your driver's release.

See also:

Write Concern (page 395) and *Write Concern Reference* (page 57).

5.2.3 Bulk Inserts

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

The `insert()` (page 961) method, when passed an array of documents, will perform a bulk insert, and inserts each document atomically. *Drivers* (page 575) provide their own interface for this kind of operation.

New in version 2.2: `insert()` (page 961) in the `mongo` (page 1066) shell gained support for bulk inserts in version 2.2.

Bulk insert can significantly increase performance by amortizing *write concern* (page 54) costs. In the drivers, you can configure write concern for batches rather than on a per-document level.

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

Note: New in version 2.0: Support for `ContinueOnError` depends on version 2.0 of the core `mongod` (page 1049) and `mongos` (page 1061) components.

If the bulk insert process generates more than one error in a batch job, the client will only receive the most recent error. All bulk operations to a *sharded collection* run with `ContinueOnError`, which applications cannot disable. See *Strategies for Bulk Inserts in Sharded Clusters* (page 539) section for more information on consideration for bulk inserts in sharded clusters.

For more information see your *driver documentation* (page 575) for details on performing bulk inserts in your application. Also consider the following resources: *Sharded Clusters* (page 57), *Strategies for Bulk Inserts in Sharded Clusters* (page 539), and *Import and Export MongoDB Data* (page 166).

5.2.4 Indexing

After every insert, update, or delete operation, MongoDB must update *every* index associated with the collection in addition to the data itself. Therefore, every index on a collection adds some amount of overhead for the performance of write operations. [4]

In general, the performance gains that indexes provide for *read operations* are worth the insertion penalty; however, when optimizing write performance, be careful when creating new indexes and always evaluate the indexes on the collection and ensure that your queries are actually using these indexes.

For more information on indexes in MongoDB consider *Indexes* (page 307) and *Indexing Strategies* (page 321).

5.2.5 Isolation

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

No other operations are atomic; however, you can attempt to isolate a write operation that affects multiple documents using the *isolation operator* (page 822).

To isolate a sequence of write operations from other read and write operations, see *Perform Two Phase Commits* (page 585).

5.2.6 Updates

Each document in a MongoDB collection has allocated *record* space which includes the entire document *and* a small amount of padding. This padding makes it possible for update operations to increase the size of a document slightly without causing the document to outgrow the allocated record size.

Documents in MongoDB can grow up to the full maximum `BSON document size` (page 1139). However, when documents outgrow their allocated record size MongoDB must allocate a new record and move the document to the new record. Update operations that do not cause a document to grow, (i.e. *in-place* updates,) are significantly more efficient than those updates that cause document growth. Use *data models* (page 107) that minimize the need for document growth when possible.

For complete examples of update operations, see *Update* (page 93).

5.2.7 Padding Factor

If an update operation does not cause the document to increase in size, MongoDB can apply the update in-place. Some updates change the size of the document, for example using the `$push` (page 818) operator to append a sub-document to an array can cause the top level document to grow beyond its allocated space.

When documents grow, MongoDB relocates the document on disk with enough contiguous space to hold the document. These relocations take longer than in-place updates, particularly if the collection has indexes that MongoDB must update all index entries. If collection has many indexes, the move will impact write throughput.

[4] The overhead for *sparse indexes* (page 314) inserts and updates to un-indexed fields is less than for non-sparse indexes. Also for non-sparse indexes, updates that don't change the record size have less indexing overhead.

To minimize document movements, MongoDB employs padding. MongoDB adaptively learns if documents in a collection tend to grow, and if they do, adds a `paddingFactor` (page 901) so that the documents have room to grow on subsequent writes. The `paddingFactor` (page 901) indicates the padding for new inserts and moves.

New in version 2.2: You can use the `collMod` (page 892) command with the `usePowerOf2Sizes` (page 892) flag so that MongoDB allocates document space in sizes that are powers of 2. This helps ensure that MongoDB can efficiently reuse the space freed as a result of deletions or document relocations. As with all padding, using document space allocations with power of 2 sizes minimizes, but does not eliminate, document movements.

To check the current `paddingFactor` (page 901) on a collection, you can run the `db.collection.stats()` (page 973) operation in the `mongo` (page 1066) shell, as in the following example:

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

Since MongoDB writes each document at a different point in time, the padding for each document will not be the same. You can calculate the padding size by subtracting 1 from the `paddingFactor` (page 901), for example:

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

For example, a `paddingFactor` (page 901) of `1.0` specifies no padding whereas a paddingFactor of `1.5` specifies a padding size of `0.5` or 50 percent (50%) of the document size.

Because the `paddingFactor` (page 901) is relative to the size of each document, you cannot calculate the exact amount of padding for a collection based on the average document size and padding factor.

If an update operation causes the document to *decrease* in size, for instance if you perform an `$unset` (page 814) or a `$pop` (page 816) update, the document remains in place and effectively has more padding. If the document remains this size, the space is not reclaimed until you perform a `compact` (page 890) or a `repairDatabase` (page 895) operation.

Note: The following operations remove padding:

- `compact` (page 890),
- `repairDatabase` (page 895), and
- initial replica sync operations.

However, with the `compact` (page 890) command, you can run the command with a `paddingFactor` or a `paddingBytes` parameter.

Padding is also removed if you use `mongoexport` (page 1093) from a collection. If you use `mongoimport` (page 1089) into a new collection, `mongoimport` (page 1089) will not add padding. If you use `mongoimport` (page 1089) with an existing collection with padding, `mongoimport` (page 1089) will not affect the existing padding.

When a database operation removes padding, subsequent update that require changes in record sizes will have reduced throughput until the collection's padding factor grows. Padding does not affect in-place, and after `compact` (page 890), `repairDatabase` (page 895), and replica set initial sync the collection will require less storage.

See also:

- *Can I manually pad documents to prevent moves during updates?* (page 744)
- The `$inc` (page 810) for in-place updates.

5.2.8 Architecture

Replica Sets

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

Large volumes of write operations, particularly bulk operations, may create situations where the secondary members have difficulty applying the replicating operations from the primary at a sufficient rate: this can cause the secondary's state to fall behind that of the primary. Secondaries that are significantly behind the primary present problems for normal operation of the replica set, particularly *failover* (page 388) in the form of *rollbacks* (page 393) as well as general *read consistency* (page 394).

To help avoid this issue, you can customize the *write concern* (page 54) to return confirmation of the write operation to another member [5] of the replica set every 100 or 1,000 operations. This provides an opportunity for secondaries to catch up with the primary. Write concern can slow the overall progress of write operations but ensure that the secondaries can maintain a largely current state with respect to the primary.

For more information on replica sets and write operations, see *Replica Acknowledged* (page 396), *Oplog Size* (page 405), and *Change the Size of the Oplog* (page 439).

Sharded Clusters

In a *sharded cluster*, MongoDB directs a given write operation to a *shard* and then performs the write on a particular *chunk* on that shard. Shards and chunks are range-based. *Shard keys* affect how MongoDB distributes documents among shards. Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster.

For more information, see *Sharded Cluster Tutorials* (page 515) and *Bulk Inserts* (page 54).

5.3 Write Concern Reference

5.3.1 Overview

Write concern is a quality of every write operation issued to a MongoDB deployment, and describes the amount of *concern* the application has for the outcome of the write operation. With weak or disabled write concern, the application can send a write operation to MongoDB and then continue without waiting for a response from the database. With stronger write concerns, write operations wait until MongoDB acknowledges or confirms a successful write operation. MongoDB provides different levels of write concern to better address the specific needs of applications.

See also:

Write Concern (page 395) for an introduction to write concern in MongoDB.

5.3.2 Available Write Concern

To provide write concern, *drivers* (page 575) issue the `getLastError` (page 861) command after a write operation and receive a document with information about the last operation. This document's `err` field contains either:

[5] Calling `getLastError` (page 861) intermittently with a `w` value of `2` or `majority` will slow the throughput of write traffic; however, this practice will allow the secondaries to remain current with the state of the primary.

- `null`, which indicates the write operations have completed successfully, or
- a description of the last error encountered.

The definition of a "successful write" depends on the arguments specified to `getLastError` (page 861), or in replica sets, the configuration of `getLastErrorDefaults` (page 477). When deciding the level of write concern for your application, see the introduction to *Write Concern* (page 395).

The `getLastError` (page 861) command has the following options to configure write concern requirements:

- `j` or "journal" option

  This option confirms that the `mongod` (page 1049) instance has written the data to the on-disk journal and ensures data is not lost if the `mongod` (page 1049) instance shuts down unexpectedly. Set to `true` to enable, as shown in the following example:

  ```
  db.runCommand( { getLastError: 1, j: "true" } )
  ```

  If you set `journal` (page 1119) to true, and the `mongod` (page 1049) does not have journaling enabled, as with `nojournal` (page 1120), then `getLastError` (page 861) will provide basic receipt acknowledgment, and will include a `jnote` field in its return document.

- `w` option

  This option provides the ability to disable write concern entirely *as well as* specifies the write concern operations for *replica sets*. See *Write Concern Considerations* (page 395) for an introduction to the fundamental concepts of write concern. By default, the `w` option is set to `1`, which provides basic receipt acknowledgment on a single `mongod` (page 1049) instance or on the *primary* in a replica set.

  The `w` option takes the following values:

  – `-1`:

    Disables all acknowledgment of write operations, and suppresses all errors, including network and socket errors.

  – `0`:

    Disables basic acknowledgment of write operations, but returns information about socket exceptions and networking errors to the application.

    Note: If you disable basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the driver will require that `mongod` (page 1049) will acknowledge the write operation.

  – `1`:

    Provides acknowledgment of write operations on a standalone `mongod` (page 1049) or the *primary* in a replica set.

  – *A number greater than 1*:

    Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. If you set `w` to a number that is greater than the number of set members that hold data, MongoDB waits for the non-existent members to become available, which means MongoDB blocks indefinitely.

  – `majority`:

    Confirms that write operations have propagated to the majority of configured replica set: a majority of the set's configured members must acknowledge the write operation before it succeeds. This ensures that write operation will *never* be subject to a rollback in the course of normal operation, and furthermore allows you to avoid hard coding assumptions about the size of your replica set into your application.

– *A tag set*:

  By specifying a *tag set* (page 444) you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.

`getLastError` (page 861) also supports a `wtimeout` setting which allows clients to specify a timeout for the write concern: if you don't specify `wtimeout` and the `mongod` (page 1049) cannot fulfill the write concern the `getLastError` (page 861) will block, potentially forever.

For more information on write concern and replica sets, see *Write Concern for Replica Sets* (page 396) for more information.

In sharded clusters, `mongos` (page 1061) instances will pass write concern on to the shard `mongod` (page 1049) instances.

CHAPTER 6

Fundamental Concepts for Document Databases

6.1 BSON Documents

MongoDB is a document-based database system, and as a result, all records, or data, in MongoDB are documents. Documents are the default representation of most user accessible data structures in the database. Documents provide structure for data in the following MongoDB contexts:

- the *records* (page 63) stored in *collections*
- the *query selectors* (page 64) that determine which records to select for read, update, and delete operations
- the *update actions* (page 65) that specify the particular field updates to perform during an update operation
- the specification of *indexes* (page 66) for collection.
- arguments to several MongoDB methods and operators, including:
  - *sort order* (page 66) for the `sort()` (page 991) method.
  - *index specification* (page 66) for the `hint()` (page 985) method.
- the output of a number of MongoDB commands and operations, including:
  - the output of `collStats` (page 900) command, and
  - the *output* (page 919) of the `serverStatus` (page 919) command.

6.1.1 Structure

The document structure in MongoDB are *BSON* objects with support for the full range of *BSON types*; however, BSON documents are conceptually, similar to *JSON* objects, and have the following structure:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

Having support for the full range of BSON types, MongoDB documents may contain field and value pairs where the value can be another document, an array, an array of documents as well as the basic types such as `Double`, `String`, and `Date`. See also *BSON Type Considerations* (page 66).

Consider the following document that contains values of varying types:

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
            }
```

The document contains the following fields:

- `_id` that holds an *ObjectId*.
- `name` that holds a *subdocument* that contains the fields `first` and `last`.
- `birth` and `death`, which both have *Date* types.
- `contribs` that holds an *array of strings*.
- `views` that holds a value of *NumberLong* type.

All field names are strings in *BSON* documents. Be aware that there are some `restrictions on field names` (page 1142) for *BSON* documents: field names cannot contain null characters, dots (`.`), or dollar signs (`$`).

Note: BSON documents may have more than one field with the same name; however, most *MongoDB Interfaces* (page 575) represent MongoDB with a structure (e.g. a hash table) that does not support duplicate field names. If you need to manipulate documents that have more than one field with the same name, see your driver's documentation for more information.

Some documents created by internal MongoDB processes may have duplicate fields, but *no* MongoDB process will *ever* add duplicate keys to an existing user document.

Type Operators

To determine the type of fields, the `mongo` (page 1066) shell provides the following operators:

- `instanceof` returns a boolean to test if a value has a specific type.
- `typeof` returns the type of a field.

Example

Consider the following operations using `instanceof` and `typeof`:

- The following operation tests whether the `_id` field is of type `ObjectId`:

  ```
  mydoc._id instanceof ObjectId
  ```

  The operation returns `true`.

- The following operation returns the type of the `_id` field:

  ```
  typeof mydoc._id
  ```

  In this case `typeof` will return the more generic `object` type rather than `ObjectId` type.

Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of a subdocument.

For example, to access an element of an array by the index position, you concatenate the array name with the dot (`.`) and zero-based index position and enclose in quotes:

```
'<array>.<index>'
```

To access a field of a subdocument with *dot-notation*, you concatenate the subdocument name with the dot (`.`) and the field name and enclose in quotes:

```
'<subdocument>.<field>'
```

See also:

- *Subdocuments* (page 43) for dot notation examples with subdocuments.
- *Arrays* (page 44) for dot notation examples with arrays.

6.1.2 Document Types in MongoDB

Record Documents

Most documents in MongoDB in *collections* store data from users' applications.

These documents have the following attributes:

- The maximum BSON document size is 16 megabytes.

  The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` (page 1110) and the documentation for your *driver* (page 575) for more information about GridFS.

- *Documents* (page 61) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

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` (page 1049) will add the `_id` field and generate the `ObjectId`.

The following document specifies a record in a collection:

```
{
  _id: 1,
  name: { first: 'John', last: 'Backus' },
  birth: new Date('Dec 03, 1924'),
  death: new Date('Mar 17, 2007'),
  contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
  awards: [
            { award: 'National Medal of Science',
              year: 1975,
              by: 'National Science Foundation' },
            { award: 'Turing Award',
              year: 1977,
              by: 'ACM' }
```

```
            ]
}
```

The document contains the following fields:

- `_id`, which must hold a unique value and is *immutable*.
- `name` that holds another *document*. This sub-document contains the fields `first` and `last`, which both hold *strings*.
- `birth` and `death` that both have *date* types.
- `contribs` that holds an *array of strings*.
- `awards` that holds an *array of documents*.

Consider the following behavior and constraints of the `_id` field in MongoDB documents:

- In documents, the `_id` field is always indexed for regular collections.
- The `_id` field may contain values of any BSON data type other than an array.

> **Warning:** To ensure functioning replication, do not store values that are of the BSON regular expression type in the `_id` field.

Consider the following options for the value of an `_id` field:

- Use an `ObjectId`. See the *ObjectId* (page 68) documentation.

  Although it is common to assign `ObjectId` values to `_id` fields, if your objects have a natural unique identifier, consider using that for the value of `_id` to save space and to avoid an additional index.

- Generate a sequence number for the documents in your collection in your application and use this value for the `_id` value. See the *Create an Auto-Incrementing Sequence Field* (page 594) tutorial for an implementation pattern.

- Generate a UUID in your application code. For a more efficient storage of the UUID values in the collection and in the `_id` index, store the UUID as a value of the BSON `BinData` type.

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

  – the binary subtype value is in the range of 0-7 or 128-135, and

  – the length of the byte array is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, or 32.

- Use your driver's BSON UUID facility to generate UUIDs. Be aware that driver implementations may implement UUID serialization and deserialization logic differently, which may not be fully compatible with other drivers. See your driver documentation for information concerning UUID interoperability.

Query Specification Documents

Query documents specify the conditions that determine which records to select for read, update, and delete operations. You can use <field>:<value> expressions to specify the equality condition and *query operator* (page 785) expressions to specify additional conditions.

When passed as an argument to methods such as the `find()` (page 951) method, the `remove()` (page 970) method, or the `update()` (page 974) method, the query document selects documents for MongoDB to return, remove, or update, as in the following:

```
db.bios.find( { _id: 1 } )
db.bios.remove( { _id: { $gt: 3 } } )
db.bios.update( { _id: 1, name: { first: 'John', last: 'Backus' } },
                <update>,
                <options> )
```

See also:

- *Query Document* (page 42) and *Read* (page 83) for more examples on selecting documents for reads.
- *Update* (page 93) for more examples on selecting documents for updates.
- *Delete* (page 101) for more examples on selecting documents for deletes.

Update Specification Documents

Update documents specify the data modifications to perform during an `update()` (page 974) operation to modify existing records in a collection. You can use *update operators* (page 810) to specify the exact actions to perform on the document fields.

Consider the update document example:

```
{
  $set: { 'name.middle': 'Warner' },
  $push: { awards: { award: 'IBM Fellow',
                     year: '1963',
                     by: 'IBM' }
         }
}
```

When passed as an argument to the `update()` (page 974) method, the update actions document:

- Modifies the field `name` whose value is another document. Specifically, the `$set` (page 814) operator updates the `middle` field in the `name` subdocument. The document uses *dot notation* (page 62) to access a field in a subdocument.
- Adds an element to the field `awards` whose value is an array. Specifically, the `$push` (page 818) operator adds another document as element to the field `awards`.

```
db.bios.update(
   { _id: 1 },
   {
     $set: { 'name.middle': 'Warner' },
     $push: { awards: {
                        award: 'IBM Fellow',
                        year: '1963',
                        by: 'IBM'
                      }
            }
   }
)
```

See also:

- *update operators* (page 810) page for the available update operators and syntax.
- *update* (page 93) for more examples on update documents.

For additional examples of updates that involve array elements, including where the elements are documents, see the `$` (page 814) positional operator.

Index Specification Documents

Index specification documents describe the fields to index on during the *index creation* (page 949). See *indexes* (page 309) for an overview of indexes. [1]

Index documents contain field and value pairs, in the following form:

```
{ field: value }
```

- `field` is the field in the documents to index.
- `value` is either 1 for ascending or -1 for descending.

The following document specifies the *multi-key index* (page 312) on the `_id` field and the `last` field contained in the subdocument `name` field. The document uses *dot notation* (page 62) to access a field in a subdocument:

```
{ _id: 1, 'name.last': 1 }
```

When passed as an argument to the `ensureIndex()` (page 949) method, the index documents specifies the index to create:

```
db.bios.ensureIndex( { _id: 1, 'name.last': 1 } )
```

Sort Order Specification Documents

Sort order documents specify the order of documents that a `query()` (page 951) returns. Pass sort order specification documents as an argument to the `sort()` (page 991) method. See the `sort()` (page 991) page for more information on sorting.

The sort order documents contain field and value pairs, in the following form:

```
{ field: value }
```

- `field` is the field by which to sort documents.
- `value` is either 1 for ascending or -1 for descending.

The following document specifies the sort order using the fields from a sub-document `name` first sort by the `last` field ascending, then by the `first` field also ascending:

```
{ 'name.last': 1, 'name.first': 1 }
```

When passed as an argument to the `sort()` (page 991) method, the sort order document sorts the results of the `find()` (page 951) method:

```
db.bios.find().sort( { 'name.last': 1, 'name.first': 1 } )
```

6.1.3 BSON Type Considerations

The following BSON types require special consideration:

ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first 4-bytes is a timestamp that reflects the ObjectId's creation. Refer to the *ObjectId* (page 68) documentation for more information.

[1] Indexes optimize a number of key *read* (page 41) and *write* (page 53) operations.

String

BSON strings are UTF-8. In general, drivers for each programming language convert from the language's string format to UTF-8 when serializing and deserializing BSON. This makes it possible to store most international characters in BSON strings with ease. [2] In addition, MongoDB `$regex` (page 798) queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 67) type. Timestamp values are a 64 bit value where:

- the first 32 bits are a `time_t` value (seconds since the Unix epoch)
- the second 32 bits are an incrementing `ordinal` for operations within a given second.

Within a single `mongod` (page 1049) instance, timestamp values are always unique.

In replication, the *oplog* has a `ts` field. The values in this field reflect the operation time, which uses a BSON timestamp value.

Note: The BSON Timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See *Date* (page 67) for more information.

If you create a BSON Timestamp using the empty constructor (e.g. `new Timestamp()`), MongoDB will only generate a timestamp *if* you use the constructor in the first field of the document. [3] Otherwise, MongoDB will generate an empty timestamp value (i.e. `Timestamp(0, 0)`.)

Changed in version 2.1: `mongo` (page 1066) shell displays the Timestamp value with the wrapper:

```
Timestamp(<time_t>, <ordinal>)
```

Prior to version 2.1, the `mongo` (page 1066) shell display the Timestamp value as a document:

```
{ t : <time_t>, i : <ordinal> }
```

Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). The official BSON specification refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed. [4] Negative values represent dates before 1970.

Consider the following examples of BSON Date:

- Construct a Date using the `new Date()` constructor in the `mongo` (page 1066) shell:

  ```
  var mydate1 = new Date()
  ```

- Construct a Date using the `ISODate()` constructor in the `mongo` (page 1066) shell:

  ```
  var mydate2 = ISODate()
  ```

[2] Given strings using UTF-8 character sets, using `sort()` (page 991) on strings will be reasonably correct; however, because internally `sort()` (page 991) uses the C++ `strcmp` api, the sort order may handle some characters incorrectly.

[3] If the first field in the document is `_id`, then you can generate a timestamp in the *second* field of a document.

[4] Prior to version 2.0, `Date` values were incorrectly interpreted as *unsigned* integers, which affected sorts, range queries, and indexes on `Date` fields. Because indexes are not recreated when upgrading, please re-index if you created an index on `Date` values with an earlier version, and dates before 1970 are relevant to your application.

- Return the `Date` value as string:

```
mydate1.toString()
```

- Return the month portion of the Date value; months are zero-indexed, so that January is month `0`:

```
mydate1.getMonth()
```

6.2 ObjectId

6.2.1 Overview

ObjectId is a 12-byte *BSON* type, constructed using:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

In MongoDB, documents stored in a collection require a unique *_id* field that acts as a *primary key*. Because ObjectIds are small, most likely unique, and fast to generate, MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified. MongoDB clients should add an `_id` field with a unique ObjectId. However, if a client does not add an `_id` field, `mongod` (page 1049) will add an `_id` field that holds an ObjectId.

Using ObjectIds for the `_id` field provides the following additional benefits:

- in the `mongo` (page 1066) shell, you can access the creation time of the `ObjectId`, using the `getTimestamp()` method.
- sorting on an `_id` field that stores `ObjectId` values is roughly equivalent to sorting by creation time.

Important: The relationship between the order of `ObjectId` values and generation time is not strict within a single second. If multiple systems, or multiple processes or threads on a single system generate values, within a single second; `ObjectId` values do not represent a strict insertion order. Clock skew between clients can also result in non-strict ordering even for values, because client drivers generate `ObjectId` values, *not* the `mongod` (page 1049) process.

Also consider the *BSON Documents* (page 61) section for related information on MongoDB's document orientation.

6.2.2 ObjectId()

The `mongo` (page 1066) shell provides the `ObjectId()` wrapper class to generate a new ObjectId, and to provide the following helper attribute and methods:

- `str`

  The hexadecimal string value of the `ObjectId()` object.

- `getTimestamp()`

  Returns the timestamp portion of the `ObjectId()` object as a Date.

- `toString()`

Returns the string representation of the `ObjectId()` object. The returned string literal has the format "`ObjectId(...)`".

Changed in version 2.2: In previous versions `ObjectId.toString()` returns the value of the ObjectId as a hexadecimal string.

- `valueOf()`

  Returns the value of the `ObjectId()` object as a hexadecimal string. The returned string is the `str` attribute.

  Changed in version 2.2: In previous versions `ObjectId.valueOf()` returns the `ObjectId()` object.

6.2.3 Examples

Consider the following uses `ObjectId()` class in the `mongo` (page 1066) 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()` 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()` 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()` method as follows:

  ```
  ObjectId("507f191e810c19729de860ea").valueOf()
  ```

This operation returns the following output:

```
507f191e810c19729de860ea
```

6.3 GridFS

GridFS is a specification for storing and retrieving files that exceed the *BSON*-document *size limit* (page 1139) of 16MB.

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, [5] and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to “skip” into the middle of a video or audio file.

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory. For more information on the indications of GridFS, see *When should I use GridFS?* (page 738).

6.3.1 Implement GridFS

To store and retrieve files using *GridFS*, use either of the following:

- A MongoDB driver. See the *drivers* (page 575) documentation for information on using GridFS with your driver.
- The `mongofiles` (page 1110) command-line tool in the `mongo` (page 1066) shell. See *mongofiles* (page 1110).

6.3.2 GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 1152).
- `files` stores the file's metadata. For details, see *The files Collection* (page 1153).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the GridFS store. Each chunk is identified by its unique *ObjectID* stored in its `_id` field.

For descriptions of all fields in the `chunks` and `files` collections, see *GridFS Reference* (page 1152).

[5] The use of the term *chunks* in the context of GridFS is not related to the use of the term *chunks* in the context of sharding.

6.3.3 GridFS Index

GridFS uses a *unique*, *compound* index on the `chunks` collection for the `files_id` and `n` fields. The `files_id` field contains the `_id` of the chunk's "parent" document. The `n` field contains the sequence number of the chunk. GridFS numbers all chunks, starting with 0. For descriptions of the documents and fields in the `chunks` collection, see *GridFS Reference* (page 1152).

The GridFS index allows efficient retrieval of chunks using the `files_id` and `n` values, as shown in the following example:

```
cursor = db.fs.chunks.find({files_id: myFileID}).sort({n:1});
```

See the relevant *driver* (page 575) documentation for the specific behavior of your GridFS application. If your driver does not create this index, issue the following operation using the `mongo` (page 1066) shell:

```
db.fs.chunks.ensureIndex( { files_id: 1, n: 1 }, { unique: true } );
```

6.3.4 Example Interface

The following is an example of the GridFS interface in Java. The example is for demonstration purposes only. For API specifics, see the relevant *driver* (page 575) documentation.

By default, the interface must support the default GridFS bucket, named `fs`, as in the following:

```
// returns default GridFS bucket (i.e. "fs" collection)
GridFS myFS = new GridFS(myDatabase);

// saves the file to "fs" GridFS bucket
myFS.createFile(new File("/tmp/largething.mpg"));
```

Optionally, interfaces may support other additional GridFS buckets as in the following example:

```
// returns GridFS bucket named "contracts"
GridFS myContracts = new GridFS(myDatabase, "contracts");

// retrieve GridFS object "smithco"
GridFSDBFile file = myContracts.findOne("smithco");

// saves the GridFS file to the file system
file.writeTo(new File("/tmp/smithco.pdf"));
```

6.4 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 1150) 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 1151) 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 575) have helper methods that form the query for the DBRef automatically. The drivers [6] 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.

6.4.1 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 1150). 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 1151).

[6] Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

6.4.2 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.

Example

DBRef document would resemble the following:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

Consider a document from a collection that stored a DBRef in a `creator` field:

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                  "$ref" : "creators",
                  "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                  "$db" : "users"
               }
}
```

The DBRef in this example, points to a document in the `creators` collection of the `users` database that has `ObjectId("5126bc054aed4daf9e2ab772")` in its `_id` field.

Note: The order of fields in the DBRef matters, and you must use the above sequence when using a DBRef.

Support

C++ The C++ driver contains no support for DBRefs. You can transverse references manually.

C# The C# driver provides access to DBRef objects with the MongoDBRef Class and supplies the FetchDBRef Method for accessing these objects.

Java The DBRef class provides supports for DBRefs from Java.

JavaScript The `mongo` (page 1066) shell's *JavaScript* (page 944) 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 1150) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

CHAPTER 7

CRUD Operations for MongoDB

These documents provide an overview and examples of common database operations, i.e. CRUD, in MongoDB.

7.1 Create

Of the four basic database operations (i.e. CRUD), *create* operations are those that add new records or *documents* to a *collection* in MongoDB. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 53); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 39) page.

7.1.1 Overview

You can create documents in a MongoDB collection using any of the following basic operations:

- *insert* (page 76)
- *updates with the upsert option* (page 81)

All insert operations in MongoDB exhibit the following properties:

- If you attempt to insert a document without the *_id* field, the client library *or* the `mongod` (page 1049) instance will add an `_id` field and populate the field with a unique *ObjectId*.
- For operations with *write concern* (page 395), if you specify an `_id` field, the `_id` field must be unique within the collection; otherwise the `mongod` (page 1049) will return a duplicate key exception.
- 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` (page 1110) and the documentation for your *driver* (page 575) for more information about GridFS.

- *Documents* (page 61) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

Note: As of these *driver versions* (page 1223), all write operations will issue a `getLastError` (page 861) command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 54) in the *Write Operations* (page 53) document for more information.

7.1.2 `insert()`

The `insert()` (page 961) is the primary method to insert a document or documents into a MongoDB collection, and has the following syntax:

```
db.collection.insert( <document> )
```

Corresponding Operation in SQL

The `insert()` (page 961) method is analogous to the `INSERT` statement.

Insert the First Document in a Collection

If the collection does not exist [1], then the `insert()` (page 961) method creates the collection during the first insert. Specifically in the example, if the collection `bios` does not exist , then the insert operation will create this collection:

```
db.bios.insert(
   {
     _id: 1,
     name: { first: 'John', last: 'Backus' },
     birth: new Date('Dec 03, 1924'),
     death: new Date('Mar 17, 2007'),
     contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
     awards: [
               {
                 award: 'W.W. McDowell Award',
                 year: 1967,
                 by: 'IEEE Computer Society'
               },
               {
                 award: 'National Medal of Science',
                 year: 1975,
```

[1] You can also view a list of the existing collections in the database using the `show collections` operation in the `mongo` (page 1066) shell.

```
                  by: 'National Science Foundation'
                },
                {
                  award: 'Turing Award',
                  year: 1977,
                  by: 'ACM'
                },
                {
                  award: 'Draper Prize',
                  year: 1993,
                  by: 'National Academy of Engineering'
                }
              ]
   }
)
```

You can confirm the insert by *querying* (page 83) the `bios` collection:

```
db.bios.find()
```

This operation returns the following document from the `bios` collection:

```
{
  "_id" : 1,
  "name" : { "first" : "John", "last" : "Backus" },
  "birth" : ISODate("1924-12-03T05:00:00Z"),
  "death" : ISODate("2007-03-17T04:00:00Z"),
  "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
  "awards" : [
                {
                  "award" : "W.W. McDowell Award",
                  "year" : 1967,
                  "by" : "IEEE Computer Society"
                },
                {
                  "award" : "National Medal of Science",
                  "year" : 1975,
                  "by" : "National Science Foundation"
                },
                {
                  "award" : "Turing Award",
                  "year" : 1977,
                  "by" : "ACM"
                },
                { "award" : "Draper Prize",
                  "year" : 1993,
                  "by" : "National Academy of Engineering"
                }
              ]
}
```

Insert a Document without Specifying an `_id` Field

If the new document does not contain an `_id` field, then the `insert()` (page 961) method adds the `_id` field to the document and generates a unique `ObjectId` for the value:

```
db.bios.insert(
   {
     name: { first: 'John', last: 'McCarthy' },
     birth: new Date('Sep 04, 1927'),
     death: new Date('Dec 24, 2011'),
     contribs: [ 'Lisp', 'Artificial Intelligence', 'ALGOL' ],
     awards: [
               {
                 award: 'Turing Award',
                 year: 1971,
                 by: 'ACM'
               },
               {
                 award: 'Kyoto Prize',
                 year: 1988,
                 by: 'Inamori Foundation'
               },
               {
                 award: 'National Medal of Science',
                 year: 1990,
                 by: 'National Science Foundation'
               }
             ]
   }
)
```

You can verify the inserted document by the querying the `bios` collection:

```
db.bios.find( { name: { first: 'John', last: 'McCarthy' } } )
```

The returned document contains an `_id` field with the generated `ObjectId` value:

```
{
  "_id" : ObjectId("50a1880488d113a4ae94a94a"),
  "name" : { "first" : "John", "last" : "McCarthy" },
  "birth" : ISODate("1927-09-04T04:00:00Z"),
  "death" : ISODate("2011-12-24T05:00:00Z"),
  "contribs" : [ "Lisp", "Artificial Intelligence", "ALGOL" ],
  "awards" : [
               {
                 "award" : "Turing Award",
                 "year" : 1971,
                 "by" : "ACM"
               },
               {
                 "award" : "Kyoto Prize",
                 "year" :1988,
                 "by" : "Inamori Foundation"
               },
               {
                 "award" : "National Medal of Science",
                 "year" : 1990,
                 "by" : "National Science Foundation"
               }
             ]
}
```

Bulk Insert Multiple Documents

If you pass an array of documents to the `insert()` (page 961) method, the `insert()` (page 961) performs a bulk insert into a collection.

The following operation inserts three documents into the `bios` collection. The operation also illustrates the *dynamic schema* characteristic of MongoDB. Although the document with `_id: 3` contains a field `title` which does not appear in the other documents, MongoDB does not require the other documents to contain this field:

```
db.bios.insert(
   [
     {
       _id: 3,
       name: { first: 'Grace', last: 'Hopper' },
       title: 'Rear Admiral',
       birth: new Date('Dec 09, 1906'),
       death: new Date('Jan 01, 1992'),
       contribs: [ 'UNIVAC', 'compiler', 'FLOW-MATIC', 'COBOL' ],
       awards: [
                  {
                    award: 'Computer Sciences Man of the Year',
                    year: 1969,
                    by: 'Data Processing Management Association'
                  },
                  {
                    award: 'Distinguished Fellow',
                    year: 1973,
                    by: ' British Computer Society'
                  },
                  {
                    award: 'W. W. McDowell Award',
                    year: 1976,
                    by: 'IEEE Computer Society'
                  },
                  {
                    award: 'National Medal of Technology',
                    year: 1991,
                    by: 'United States'
                  }
                ]
     },
     {
       _id: 4,
       name: { first: 'Kristen', last: 'Nygaard' },
       birth: new Date('Aug 27, 1926'),
       death: new Date('Aug 10, 2002'),
       contribs: [ 'OOP', 'Simula' ],
       awards: [
                  {
                    award: 'Rosing Prize',
                    year: 1999,
                    by: 'Norwegian Data Association'
                  },
                  {
                    award: 'Turing Award',
                    year: 2001,
                    by: 'ACM'
                  },
                  {
```

```
                  award: 'IEEE John von Neumann Medal',
                  year: 2001,
                  by: 'IEEE'
                }
              ]
     },
     {
       _id: 5,
       name: { first: 'Ole-Johan', last: 'Dahl' },
       birth: new Date('Oct 12, 1931'),
       death: new Date('Jun 29, 2002'),
       contribs: [ 'OOP', 'Simula' ],
       awards: [
                {
                  award: 'Rosing Prize',
                  year: 1999,
                  by: 'Norwegian Data Association'
                },
                {
                  award: 'Turing Award',
                  year: 2001,
                  by: 'ACM'
                },
                {
                  award: 'IEEE John von Neumann Medal',
                  year: 2001,
                  by: 'IEEE'
                }
              ]
     }
   ]
)
```

Insert a Document with `save()`

The `save()` (page 972) method performs an insert if the document to save does not contain the `_id` field.

The following `save()` (page 972) operation performs an insert into the `bios` collection since the document does not contain the `_id` field:

```
db.bios.save(
   {
        name: { first: 'Guido', last: 'van Rossum'},
        birth: new Date('Jan 31, 1956'),
        contribs: [ 'Python' ],
        awards: [
                  {
                    award: 'Award for the Advancement of Free Software',
                    year: 2001,
                    by: 'Free Software Foundation'
                  },
                  {
                    award: 'NLUUG Award',
                    year: 2003,
                    by: 'NLUUG'
                  }
                ]
```

```
    }
)
```

7.1.3 `update()` Operations with the `upsert` Flag

The `update()` (page 974) operation in MongoDB accepts an "`upsert`" flag that modifies the behavior of `update()` (page 974) from *updating existing documents* (page 93), to inserting data.

These `update()` (page 974) operations with the upsert flag eliminate the need to perform an additional operation to check for existence of a record before performing either an update or an insert operation. These update operations have the use `<query>` argument to determine the write operation:

- If the query matches an existing document(s), the operation is an *update* (page 93).
- If the query matches no document in the collection, the operation is an *insert* (page 75).

An upsert operation has the following syntax [2]:

```
db.collection.update( <query>,
                      <update>,
                      { upsert: true } )
```

Insert a Document that Contains `field` and `value` Pairs

If no document matches the `<query>` argument, the `upsert` performs an insert. If the `<update>` argument includes only field and value pairs, the new document contains the fields and values specified in the `<update>` argument. If query does not include an `_id` field, the operation adds the `_id` field and generates a unique `ObjectId` for its value.

The following update inserts a new document into the `bios` collection [2]:

```
db.bios.update(
   { name: { first: 'Dennis', last: 'Ritchie'} },
   {
     name: { first: 'Dennis', last: 'Ritchie'},
     birth: new Date('Sep 09, 1941'),
     death: new Date('Oct 12, 2011'),
     contribs: [ 'UNIX', 'C' ],
     awards: [
               {
                 award: 'Turing Award',
                 year: 1983,
                 by: 'ACM'
               },
               {
                 award: 'National Medal of Technology',
                 year: 1998,
                 by: 'United States'
               },
               {
                 award: 'Japan Prize',
                 year: 2011,
                 by: 'The Japan Prize Foundation'
               }
```

[2] Prior to version 2.2, in the `mongo` (page 1066) shell, you would specify the `upsert` and the `multi` options in the `update()` (page 974) method as positional boolean options. See `update()` (page 974) for details.

```
                ]
    },
    { upsert: true }
)
```

Insert a Document that Contains Update Operator Expressions

If no document matches the <query> argument, the update operation inserts a new document. If the <update> argument includes only *update operators* (page 810), the new document contains the fields and values from <query> argument with the operations from the <update> argument applied.

The following operation inserts a new document into the bios collection [2]:

```
db.bios.update(
    {
      _id: 7,
      name: { first: 'Ken', last: 'Thompson' }
    },
    {
      $set: {
               birth: new Date('Feb 04, 1943'),
               contribs: [ 'UNIX', 'C', 'B', 'UTF-8' ],
               awards: [
                           {
                             award: 'Turing Award',
                             year: 1983,
                             by: 'ACM'
                           },
                           {
                             award: 'IEEE Richard W. Hamming Medal',
                             year: 1990,
                             by: 'IEEE'
                           },
                           {
                             award: 'National Medal of Technology',
                             year: 1998,
                             by: 'United States'
                           },
                           {
                             award: 'Tsutomu Kanai Award',
                             year: 1999,
                             by: 'IEEE'
                           },
                           {
                             award: 'Japan Prize',
                             year: 2011,
                             by: 'The Japan Prize Foundation'
                           }
                         ]
             }
    },
    { upsert: true }
)
```

Update operations with `save()`

The `save()` (page 972) method is identical to an *update operation with the upsert flag* (page 81)

performs an upsert if the document to save contains the `_id` field. To determine whether to perform an insert or an update, `save()` (page 972) method queries documents on the `_id` field.

The following operation performs an upsert that inserts a document into the `bios` collection since no documents in the collection contains an `_id` field with the value `10`:

```
db.bios.save(
   {
     _id: 10,
     name: { first: 'Yukihiro', aka: 'Matz', last: 'Matsumoto'},
     birth: new Date('Apr 14, 1965'),
     contribs: [ 'Ruby' ],
     awards: [
                 {
                   award: 'Award for the Advancement of Free Software',
                   year: '2011',
                   by: 'Free Software Foundation'
                 }
               ]
   }
)
```

7.2 Read

Of the four basic database operations (i.e. CRUD), read operations are those that retrieve records or *documents* from a *collection* in MongoDB. For general information about read operations and the factors that affect their performance, see *Read Operations* (page 41); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 39) page.

7.2.1 Overview

You can retrieve documents from MongoDB using either of the following methods:

- *find* (page 84)
- *findOne* (page 92)

7.2.2 `find()`

The `find()` (page 951) method is the primary method to select documents from a collection. The `find()` (page 951) method returns a cursor that contains a number of documents. Most *drivers* (page 575) provide application developers with a native iterable interface for handling cursors and accessing documents. The `find()` (page 951) method has the following syntax:

```
db.collection.find( <query>, <projection> )
```

Corresponding Operation in SQL

The `find()` (page 951) method is analogous to the `SELECT` statement, while:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<projection>` argument corresponds to the list of fields to select from the result set.

The examples refer to a collection named `bios` that contains documents with the following prototype:

```
{
  "_id" : 1,
  "name" : {
             "first" : "John",
             "last" :"Backus"
           },
  "birth" : ISODate("1924-12-03T05:00:00Z"),
  "death" : ISODate("2007-03-17T04:00:00Z"),
  "contribs" : [ "Fortran", "ALGOL", "Backus-Naur Form", "FP" ],
  "awards" : [
              {
                "award" : "W.W. McDowellAward",
                "year" : 1967,
                "by" : "IEEE Computer Society"
              },
              {
                "award" : "National Medal of Science",
                "year" : 1975,
                "by" : "National Science Foundation"
              },
              {
                "award" : "Turing Award",
                "year" : 1977,
                "by" : "ACM"
              },
              {
                "award" : "Draper Prize",
                "year" : 1993,
                "by" : "National Academy of Engineering"
              }
  ]
}
```

Note: In the `mongo` (page 1066) shell, you can format the output by adding `.pretty()` to the `find()` (page 951) method call.

Return All Documents in a Collection

If there is no `<query>` argument, the `find()` (page 951) method selects all documents from a collection.

The following operation returns all documents (or more precisely, a cursor to all documents) in the `bios` collection:

```
db.bios.find()
```

Return Documents that Match Query Conditions

If there is a <query> argument, the `find()` (page 951) method selects all documents from a collection that satisfies the query specification.

Equality Matches

The following operation returns a cursor to documents in the `bios` collection where the field `_id` equals `5`:

```
db.bios.find(
   {
     _id: 5
   }
)
```

Using Operators

The following operation returns a cursor to all documents in the `bios` collection where the field `_id` equals `5` or `ObjectId("507c35dd8fada716c89d0013")`:

```
db.bios.find(
   {
     _id: { $in: [ 5,  ObjectId("507c35dd8fada716c89d0013") ] }
   }
)
```

Query for Ranges

You may combine comparison operators to specify ranges:

```
db.collection.find( { field: { $gt: value1, $lt: value2 } } );
```

This statement returns all documents with `field` between `value1` and `value2`.

Note: If the field contains an array and the query has multiple conditional operators, the field as a whole will match if either a single array element meets the conditions or a combination of array elements meet the conditions.

Example

Query a field that contains an array.

A collection `students` contains the following documents where the `score` field contains an array of values:

```
{ "_id" : 1, "score" : [ -1, 3 ] }
{ "_id" : 2, "score" : [ 1, 5 ] }
{ "_id" : 3, "score" : [ 5, 5 ] }
```

Then, the following query:

```
db.students.find( { score: { $gt: 0, $lt: 2 } } )
```

Will match the following documents:

```
{ "_id" : 1, "score" : [ -1, 3 ] }
{ "_id" : 2, "score" : [ 1, 5 ] }
```

- In the document with `_id` equal to `1`, the `score: [ -1, 3 ]` as a whole meets the specified conditions since the element `-1` meets the `$lt: 2` condition and the element `3` meets the `$gt: 0` condition.
- In the document with `_id` equal to `2`, the `score: [ 1, 5 ]` as a whole meets the specified conditions since the element `1` meets both the `$lt: 2` condition and the `$gt: 0` condition.

On Arrays

Query an Element The following operation returns a cursor to all documents in the `bios` collection where the array field `contribs` contains the element `'UNIX'`:

```
db.bios.find(
   {
     contribs: 'UNIX'
   }
)
```

Query Multiple Fields on an Array of Documents The following operation returns a cursor to all documents in the `bios` collection where `awards` array contains a subdocument element that contains the `award` field equal to `'Turing Award'` and the `year` field greater than 1980:

```
db.bios.find(
   {
      awards: {
                $elemMatch: {
                     award: 'Turing Award',
                     year: { $gt: 1980 }
                }
      }
   }
)
```

On Subdocuments

Exact Matches The following operation returns a cursor to all documents in the `bios` collection where the subdocument `name` is *exactly* `{ first: 'Yukihiro', last: 'Matsumoto' }`, including the order:

```
db.bios.find(
    {
      name: {
              first: 'Yukihiro',
              last: 'Matsumoto'
            }
    }
)
```

The `name` field must match the sub-document exactly, including order. For instance, the query would **not** match documents with `name` fields that held either of the following values:

```
{
   first: 'Yukihiro',
   aka: 'Matz',
   last: 'Matsumoto'
}
```

```
{
   last: 'Matsumoto',
   first: 'Yukihiro'
}
```

Fields of a Subdocument The following operation returns a cursor to all documents in the `bios` collection where the subdocument `name` contains a field `first` with the value `'Yukihiro'` and a field `last` with the value `'Matsumoto'`; the query uses *dot notation* to access fields in a subdocument:

```
db.bios.find(
   {
     'name.first': 'Yukihiro',
     'name.last': 'Matsumoto'
   }
)
```

The query matches the document where the `name` field contains a subdocument with the field `first` with the value `'Yukihiro'` and a field `last` with the value `'Matsumoto'`. For instance, the query would match documents with `name` fields that held either of the following values:

```
{
  first: 'Yukihiro',
  aka: 'Matz',
  last: 'Matsumoto'
}

{
  last: 'Matsumoto',
  first: 'Yukihiro'
}
```

Logical Operators

`OR` Disjunctions The following operation returns a cursor to all documents in the `bios` collection where either the field `first` in the sub-document `name` starts with the letter `G` **or** where the field `birth` is less than `new Date('01/01/1945')`:

```
db.bios.find(
   { $or: [
            { 'name.first' : /^G/ },
            { birth: { $lt: new Date('01/01/1945') } }
          ]
   }
)
```

`AND` Conjunctions The following operation returns a cursor to all documents in the `bios` collection where the field `first` in the subdocument `name` starts with the letter `K` **and** the array field `contribs` contains the element `UNIX`:

```
db.bios.find(
   {
      'name.first': /^K/,
      contribs: 'UNIX'
   }
)
```

In this query, the parameters (i.e. the selections of both fields) combine using an implicit logical AND for criteria on different fields `contribs` and `name.first`. For multiple `AND` criteria on the same field, use the `$and` (page 791) operator.

With a Projection

If there is a `<projection>` argument, the `find()` (page 951) method returns only those fields as specified in the `<projection>` argument to include or exclude:

Note: The `_id` field is implicitly included in the `<projection>` argument. In projections that explicitly include fields, `_id` is the only field that you can explicitly exclude. Otherwise, you cannot mix include field and exclude field specifications.

Specify the Fields to Return

The following operation finds all documents in the `bios` collection and returns only the `name` field, the `contribs` field, and the `_id` field:

```
db.bios.find(
    { },
    { name: 1, contribs: 1 }
 )
```

Explicitly Exclude the `_id` Field

The following operation finds all documents in the `bios` collection and returns only the `name` field and the `contribs` field:

```
db.bios.find(
   { },
   { name: 1, contribs: 1, _id: 0 }
)
```

Return All but the Excluded Fields

The following operation finds the documents in the `bios` collection where the `contribs` field contains the element `'OOP'` and returns all fields *except* the `_id` field, the `first` field in the `name` subdocument, and the `birth` field from the matching documents:

```
db.bios.find(
   { contribs: 'OOP' },
   { _id: 0, 'name.first': 0, birth: 0 }
)
```

On Arrays and Subdocuments

The following operation finds all documents in the `bios` collection and returns the `last` field in the `name` subdocument and the first two elements in the `contribs` array:

```
db.bios.find(
   { },
   {
     _id: 0,
     'name.last': 1,
     contribs: { $slice: 2 }
   }
)
```

See also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 44) for more examples on accessing arrays.
- *Subdocuments* (page 43) for more examples on accessing subdocuments.
- `$elemMatch` (page 809) query operator for more information on matching array elements.
- `$elemMatch` (page 824) projection operator for additional information on restricting array elements to return.

Iterate the Returned Cursor

The `find()` (page 951) method returns a *cursor* to the results; however, in the `mongo` (page 1066) shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times [3] to print up to the first 20 documents that match the query, as in the following example:

```
db.bios.find( );
```

With Variable Name

When you assign the `find()` (page 951) to a variable, you can type the name of the cursor variable to iterate up to 20 times [1] and print the matching documents, as in the following example:

```
var myCursor = db.bios.find( );

myCursor
```

With `next()` Method

You can use the cursor method `next()` (page 989) to access the documents, as in the following example:

```
var myCursor = db.bios.find( );

var myDocument = myCursor.hasNext() ? myCursor.next() : null;

if (myDocument) {
    var myName = myDocument.name;
    print (tojson(myName));
}
```

To print, you can also use the `printjson()` method instead of `print(tojson())`:

[3] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Cursor Flags* (page 51) and *Cursor Behaviors* (page 50) for more information.

```
if (myDocument) {
   var myName = myDocument.name;
   printjson(myName);
}
```

With `forEach()` Method

You can use the cursor method `forEach()` (page 984) to iterate the cursor and access the documents, as in the following example:

```
var myCursor = db.bios.find( );

myCursor.forEach(printjson);
```

For more information on cursor handling, see:

- `cursor.hasNext()` (page 985)
- `cursor.next()` (page 989)
- `cursor.forEach()` (page 984)
- *cursors* (page 49)
- *JavaScript cursor methods* (page 977)

Modify the Cursor Behavior

In addition to the `<query>` and the `<projection>` arguments, the `mongo` (page 1066) shell and the *drivers* (page 575) provide several cursor methods that you can call on the *cursor* returned by `find()` (page 951) method to modify its behavior, such as:

Order Documents in the Result Set

The `sort()` (page 991) method orders the documents in the result set.

The following operation returns all documents (or more precisely, a cursor to all documents) in the `bios` collection ordered by the `name` field ascending:

```
db.bios.find().sort( { name: 1 } )
```

`sort()` (page 991) corresponds to the `ORDER BY` statement in SQL.

Limit the Number of Documents to Return

The `limit()` (page 985) method limits the number of documents in the result set.

The following operation returns at most `5` documents (or more precisely, a cursor to at most 5 documents) in the `bios` collection:

```
db.bios.find().limit( 5 )
```

`limit()` (page 985) corresponds to the `LIMIT` statement in SQL.

Set the Starting Point of the Result Set

The `skip()` (page 990) method controls the starting point of the results set.

The following operation returns all documents, skipping the first 5 documents in the `bios` collection:

```
db.bios.find().skip( 5 )
```

Combine Cursor Methods

You can chain these cursor methods, as in the following examples [4]:

```
db.bios.find().sort( { name: 1 } ).limit( 5 )
db.bios.find().limit( 5 ).sort( { name: 1 } )
```

See the *JavaScript cursor methods* (page 977) reference and your *driver* (page 575) documentation for additional references. See *Cursors* (page 49) for more information regarding cursors.

7.2.3 `findOne()`

The `findOne()` (page 955) method selects a single document from a collection and returns that document. `findOne()` (page 955) does *not* return a cursor.

The `findOne()` (page 955) method has the following syntax:

```
db.collection.findOne( <query>, <projection> )
```

Except for the return value, `findOne()` (page 955) method is quite similar to the `find()` (page 951) method; in fact, internally, the `findOne()` (page 955) method is the `find()` (page 951) method with a limit of 1.

With Empty Query Specification

If there is no `<query>` argument, the `findOne()` (page 955) method selects just one document from a collection.

The following operation returns a single document from the `bios` collection:

```
db.bios.findOne()
```

With a Query Specification

If there is a `<query>` argument, the `findOne()` (page 955) method selects the first document from a collection that meets the `<query>` argument:

The following operation returns the first matching document from the `bios` collection where either the field `first` in the subdocument `name` starts with the letter `G` **or** where the field `birth` is less than `new Date('01/01/1945')`:

```
db.bios.findOne(
   {
     $or: [
            { 'name.first' : /^G/ },
            { birth: { $lt: new Date('01/01/1945') } }
```

[4] Regardless of the order you chain the `limit()` (page 985) and the `sort()` (page 991), the request to the server has the structure that treats the query and the `sort()` (page 991) modifier as a single object. Therefore, the `limit()` (page 985) operation method is always applied after the `sort()` (page 991) regardless of the specified order of the operations in the chain. See the *meta query operators* (page 827) for more information.

```
          ]
    }
)
```

With a Projection

You can pass a <projection> argument to `findOne()` (page 955) to control the fields included in the result set.

Specify the Fields to Return

The following operation finds a document in the `bios` collection and returns only the `name` field, the `contribs` field, and the `_id` field:

```
db.bios.findOne(
    { },
    { name: 1, contribs: 1 }
)
```

Return All but the Excluded Fields

The following operation returns a document in the `bios` collection where the `contribs` field contains the element `OOP` and returns all fields *except* the `_id` field, the `first` field in the `name` subdocument, and the `birth` field from the matching documents:

```
db.bios.findOne(
   { contribs: 'OOP' },
   { _id: 0, 'name.first': 0, birth: 0 }
)
```

Access the `findOne` Result

Although similar to the `find()` (page 951) method, because the `findOne()` (page 955) method returns a document rather than a cursor, you cannot apply the cursor methods such as `limit()` (page 985), `sort()` (page 991), and `skip()` (page 990) to the result of the `findOne()` (page 955) method. However, you can access the document directly, as in the example:

```
var myDocument = db.bios.findOne();

if (myDocument) {
   var myName = myDocument.name;

   print (tojson(myName));
}
```

7.3 Update

Of the four basic database operations (i.e. CRUD), *update* operations are those that modify existing records or *documents* in a MongoDB *collection*. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 53); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 39) page.

7.3.1 Overview

Update operation modifies an existing *document* or documents in a *collection*. MongoDB provides the following methods to perform update operations:

- *update* (page 94)
- *save* (page 98)

Note: Consider the following behaviors of MongoDB's update operations.

- When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk and may reorder the document fields depending on the type of update.
- As of these *driver versions* (page 1223), all write operations will issue a `getLastError` (page 861) command to confirm the result of the write operation:

  ```
  { getLastError: 1 }
  ```

  Refer to the documentation on *write concern* (page 54) in the *Write Operations* (page 53) document for more information.

7.3.2 Update

The `update()` (page 974) method is the primary method used to modify documents in a MongoDB collection. By default, the `update()` (page 974) method updates a **single** document, but by using the `multi` option, `update()`

(page 974) can update all documents that match the query criteria in the collection. The `update()` (page 974) method can either replace the existing document with the new document or update specific fields in the existing document.

The `update()` (page 974) has the following syntax [5]:

```
db.collection.update( <query>, <update>, <options> )
```

Corresponding operation in SQL

The `update()` (page 974) method corresponds to the `UPDATE` operation in SQL, and:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<update>` corresponds to the `SET ...` statement.

The default behavior of the `update()` (page 974) method updates a **single** document and would correspond to the SQL `UPDATE` statement with the `LIMIT 1`. With the `multi` option, `update()` (page 974) method would correspond to the SQL `UPDATE` statement without the `LIMIT` clause.

Modify with Update Operators

If the `<update>` argument contains only *update operator* (page 100) expressions such as the `$set` (page 814) operator expression, the `update()` (page 974) method updates the corresponding fields in the document. To update fields in subdocuments, MongoDB uses *dot notation*.

Update a Field in a Document

Use `$set` (page 814) to update a value of a field.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and sets the value of the field `middle`, in the subdocument `name`, to `Warner`:

```
db.bios.update(
   { _id: 1 },
   {
     $set: { 'name.middle': 'Warner' },
   }
)
```

Add a New Field to a Document

If the `<update>` argument contains fields not currently in the document, the `update()` (page 974) method adds the new fields to the document.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and adds to that document a new `mbranch` field and a new `aka` field in the subdocument `name`:

```
db.bios.update(
   { _id: 3 },
   { $set: {
              mbranch: 'Navy',
              'name.aka': 'Amazing Grace'
```

[5] This examples uses the interface added in MongoDB 2.2 to specify the `multi` and the `upsert` options in a document form.
Prior to version 2.2, in the `mongo` (page 1066) shell, you would specify the `upsert` and the `multi` options in the `update()` (page 974) method as positional boolean options. See `update()` (page 974) for details.

```
            }
    }
)
```

Remove a Field from a Document

If the <update> argument contains `$unset` (page 814) operator, the `update()` (page 974) method removes the field from the document.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and removes the `birth` field from the document:

```
db.bios.update(
   { _id: 3 },
   { $unset: { birth: 1 } }
)
```

Update Arrays

Update an Element by Specifying Its Position If the update operation requires an update of an element in an array field, the `update()` (page 974) method can perform the update using the position of the element and *dot notation*. Arrays in MongoDB are zero-based.

The following operation queries the `bios` collection for the first document with `_id` field equal to `1` and updates the second element in the `contribs` array:

```
db.bios.update(
   { _id: 1 },
   { $set: { 'contribs.1': 'ALGOL 58' } }
)
```

Update an Element without Specifying Its Position The `update()` (page 974) method can perform the update using the `$` (page 814) positional operator if the position is not known. The array field must appear in the `query` argument in order to determine which array element to update.

The following operation queries the `bios` collection for the first document where the `_id` field equals `3` and the `contribs` array contains an element equal to `compiler`. If found, the `update()` (page 974) method updates the first matching element in the array to `A compiler` in the document:

```
db.bios.update(
        { _id: 3, 'contribs': 'compiler' },
        { $set: { 'contribs.$': 'A compiler' } }
    )
```

Update a Document Element without Specifying Its Position The `update()` (page 974) method can perform the update of an array that contains subdocuments by using the positional operator (i.e. `$` (page 814)) and the *dot notation*.

The following operation queries the `bios` collection for the first document where the `_id` field equals `6` and the `awards` array contains a subdocument element with the `by` field equal to `ACM`. If found, the `update()` (page 974) method updates the `by` field in the first matching subdocument:

```
db.bios.update(
   { _id: 6, 'awards.by': 'ACM'  } ,
   { $set: { 'awards.$.by': 'Association for Computing Machinery' } }
)
```

Add an Element to an Array The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and adds a new element to the `awards` field:

```
db.bios.update(
   { _id: 1 },
   {
     $push: { awards: { award: 'IBM Fellow', year: 1963, by: 'IBM' } }
   }
)
```

Update Multiple Documents

If the `<options>` argument contains the `multi` option set to `true` or `1`, the `update()` (page 974) method updates all documents that match the query.

The following operation queries the `bios` collection for all documents where the `awards` field contains a subdocument element with the `award` field equal to `Turing` and sets the `turing` field to `true` in the matching documents [6]:

```
db.bios.update(
   { 'awards.award': 'Turing' },
   { $set: { turing: true } },
   { multi: true }
)
```

Replace Existing Document with New Document

If the `<update>` argument contains only field and value pairs, the `update()` (page 974) method *replaces* the existing document with the document in the `<update>` argument, except for the `_id` field.

The following operation queries the `bios` collection for the first document that has a `name` field equal to `{ first: 'John', last: 'McCarthy' }` and replaces all but the `_id` field in the document with the fields in the `<update>` argument:

```
db.bios.update(
   { name: { first: 'John', last: 'McCarthy' } },
   { name: { first: 'Ken', last: 'Iverson' },
     born: new Date('Dec 17, 1941'),
     died: new Date('Oct 19, 2004'),
     contribs: [ 'APL', 'J' ],
     awards: [
               { award: 'Turing Award',
                 year: 1979,
                 by: 'ACM' },
               { award: 'Harry H. Goode Memorial Award',
                 year: 1975,
                 by: 'IEEE Computer Society' },
```

[6] Prior to version 2.2, in the `mongo` (page 1066) shell, you would specify the `upsert` and the `multi` options in the `update()` (page 974) method as positional boolean options. See `update()` (page 974) for details.

```
                { award: 'IBM Fellow',
                  year: 1970,
                  by: 'IBM' }
              ]
   }
)
```

7.3.3 `update()` Operations with the `upsert` Flag

If you set the `upsert` option in the `<options>` argument to `true` or `1` and no existing document match the `<query>` argument, the `update()` (page 974) method can insert a new document into the collection. [7]

The following operation queries the `bios` collection for a document with the `_id` field equal to `11` and the `name` field equal to `{ first: 'James', last: 'Gosling'}`. If the query selects a document, the operation performs an update operation. If a document is not found, `update()` (page 974) inserts a new document containing the fields and values from `<query>` argument with the operations from the `<update>` argument applied. [8]

```
db.bios.update(
   { _id:11, name: { first: 'James', last: 'Gosling' } },
   {
     $set: {
             born: new Date('May 19, 1955'),
             contribs: [ 'Java' ],
             awards: [
                       {
                         award: 'The Economist Innovation Award',
                         year: 2002,
                         by: 'The Economist'
                       },
                       {
                         award: 'Officer of the Order of Canada',
                         year: 2007,
                         by: 'Canada'
                       }
                     ]
           }
   },
   { upsert: true }
)
```

See also *Update Operations with the Upsert Flag* (page 81) in the *Create* (page 75) document.

7.3.4 Save

The `save()` (page 972) method performs a special type of `update()` (page 974), depending on the `_id` field of the specified document.

The `save()` (page 972) method has the following syntax:

```
db.collection.save( <document> )
```

[7] Prior to version 2.2, in the `mongo` (page 1066) shell, you would specify the `upsert` and the `multi` options in the `update()` (page 974) method as positional boolean options. See `update()` (page 974) for details.

[8] If the `<update>` argument includes only field and value pairs, the new document contains the fields and values specified in the `<update>` argument. If the `<update>` argument includes only *update operators* (page 100), the new document contains the fields and values from `<query>` argument with the operations from the `<update>` argument applied.

Behavior

If you specify a document with an _id field, save() (page 972) performs an update() (page 974) with the upsert option set: if an existing document in the collection has the same _id, save() (page 972) updates that document, and inserts the document otherwise. If you do not specify a document with an _id field to save() (page 972), performs an insert() (page 961) operation.

That is, save() (page 972) method is equivalent to the update() (page 974) method with the upsert option and a <query> argument with an _id field.

Example

Consider the following pseudocode explanation of save() (page 972) as an illustration of its behavior:

```
function save( doc ) {
  if( doc["_id"] ) {
       update( {_id: doc["_id"] }, doc, { upsert: true } );
    }
  else {
       insert(doc);
  }
}
```

Save Performs an Update

If the <document> argument contains the _id field that exists in the collection, the save() (page 972) method performs an update that replaces the existing document with the <document> argument.

The following operation queries the bios collection for a document where the _id equals ObjectId("507c4e138fada716c89d0014") and replaces the document with the <document> argument:

```
db.bios.save(
   {
     _id: ObjectId("507c4e138fada716c89d0014"),
     name: { first: 'Martin', last: 'Odersky' },
     contribs: [ 'Scala' ]
   }
)
```

See also:

Insert a Document with save() (page 80) and *Update operations with save()* (page 83) in the *Create* (page 75) section.

7.3.5 Update Operators

Fields

Name	Description
`$inc` (page 810)	Increments the value of the field by the specified amount.
`$rename` (page 810)	Renames a field.
`$setOnInsert` (page 813)	Sets the value of a field upon documentation creation during an upsert. Has no effect on update operations that modify existing documents.
`$set` (page 814)	Sets the value of a field in an existing document.
`$unset` (page 814)	Removes the specified field from an existing document.

Array

Operators

Name	Description
`$` (page 814)	Acts as a placeholder to update the first element that matches the query condition in an update.
`$addToSet` (page 815)	Adds elements to an existing array only if they do not already exist in the set.
`$pop` (page 816)	Removes the first or last item of an array.
`$pullAll` (page 816)	Removes multiple values from an array.
`$pull` (page 817)	Removes items from an array that match a query statement.
`$pushAll` (page 817)	*Deprecated.* Adds several items to an array.
`$push` (page 818)	Adds an item to an array.

Modifiers

Name	Description
`$each` (page 819)	Modifies the `$push` (page 818) and `$addToSet` (page 815) operators to append multiple items for array updates.
`$slice` (page 819)	Modifies the `$push` (page 818) operator to limit the size of updated arrays.
`$sort` (page 820)	Modifies the `$push` (page 818) operator to reorder documents stored in an array.

Bitwise

Name	Description
`$bit` (page 821)	Performs bitwise `AND` and `OR` updates of integer values.

Isolation

Name	Description
`$isolated` (page 822)	Modifies behavior of multi-updates to improve the isolation of the operation.

7.4 Delete

Of the four basic database operations (i.e. CRUD), *delete* operations are those that remove documents from a *collection* in MongoDB.

For general information about write operations and the factors that affect their performance, see *Write Operations* (page 53); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 39) page.

7.4.1 Overview

The *remove()* (page 101) method in the `mongo` (page 1066) shell provides this operation, as do corresponding methods in the *drivers* (page 575).

Note: As of these *driver versions* (page 1223), all write operations will issue a `getLastError` (page 861) command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 54) in the *Write Operations* (page 53) document for more information.

Use the `remove()` (page 970) method to delete documents from a collection. The `remove()` (page 970) method has the following syntax:

```
db.collection.remove( <query>, <justOne> )
```

Corresponding operation in SQL

The `remove()` (page 970) method is analogous to the `DELETE` statement, and:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<justOne>` argument takes a Boolean and has the same effect as `LIMIT 1`.

`remove()` (page 970) deletes documents from the collection. If you do not specify a query, `remove()` (page 970) removes all documents from a collection, but does not remove the indexes. [9]

Note: For large deletion operations, it may be more efficient to copy the documents that you want to keep to a new

[9] To remove all documents from a collection, it may be more efficient to use the `drop()` (page 948) method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

collection and then use `drop()` (page 948) on the original collection.

7.4.2 Remove All Documents that Match a Condition

If there is a `<query>` argument, the `remove()` (page 970) method deletes from the collection all documents that match the argument.

The following operation deletes all documents from the `bios` collection where the subdocument `name` contains a field `first` whose value starts with `G`:

```
db.bios.remove( { 'name.first' : /^G/ } )
```

7.4.3 Remove a Single Document that Matches a Condition

If there is a `<query>` argument and you specify the `<justOne>` argument as `true` or `1`, `remove()` (page 970) only deletes a single document from the collection that matches the query.

The following operation deletes a single document from the `bios` collection where the `turing` field equals `true`:

```
db.bios.remove( { turing: true }, 1 )
```

7.4.4 Remove All Documents from a Collection

If there is no `<query>` argument, the `remove()` (page 970) method deletes all documents from a collection. The following operation deletes all documents from the `bios` collection:

```
db.bios.remove()
```

Note: This operation is not equivalent to the `drop()` (page 948) method.

7.4.5 Capped Collection

You cannot use the `remove()` (page 970) method with a *capped collection*.

7.4.6 Isolation

If the `<query>` argument to the `remove()` (page 970) 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 `$isolated` (page 822) isolation operator, effectively isolating the delete operation from other write operations. To isolate the operation, include `$isolated: 1` in the `<query>` parameter as in the following example:

```
db.bios.remove( { turing: true, $isolated: 1 } )
```

Part III

Data Modeling

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Although you may be able to use different structures for a single data set in MongoDB, different data models may have significant impacts on MongoDB and application performance. Consider *Data Modeling Considerations for MongoDB Applications* (page 107) for a conceptual overview of data modeling problems in MongoDB, and the *Data Modeling Patterns* (page 113) documents for examples of different approaches to data models.

See also:

Use Cases (page 629) for overviews of application design, including data models, with MongoDB.

CHAPTER 8

Background

8.1 Data Modeling Considerations for MongoDB Applications

8.1.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. This means that:

- documents in the same collection do not need to have the same set of fields or structure, and
- common fields in a collection's documents may hold different types of data.

Each document only needs to contain relevant fields to the entity or object that the document represents. In practice, *most* documents in a collection share a similar structure. Schema flexibility means that you can model your documents in MongoDB so that they can closely resemble and reflect application-level objects.

As in all data modeling, when developing data models (i.e. *schema designs*) for MongoDB, you must consider the inherent properties and requirements of the application objects and the relationships between application objects. MongoDB data models must also reflect:

- how data will grow and change over time, and
- the kinds of queries your application will perform.

These considerations and requirements force developers to make a number of multi-factored decisions when modeling data, including:

- normalization and de-normalization.

  These decisions reflect the degree to which the data model should store related pieces of data in a single document. Fully normalized data models describe relationships using *references* (page 1150) between documents, while de-normalized models may store redundant information across related models.

- *indexing strategy* (page 321).
- representation of data in arrays in *BSON*.

Although a number of data models may be functionally equivalent for a given application, different data models may have significant impacts on MongoDB and applications performance.

This document provides a high level overview of these data modeling decisions and factors. In addition, consider the *Data Modeling Patterns and Examples* (page 111) section which provides more concrete examples of all the discussed patterns.

8.1.2 Data Modeling Decisions

Data modeling decisions involve determining how to structure the documents to model the data effectively. The primary decision is whether to *embed* (page 108) or to *use references* (page 108).

Embedding

To de-normalize data, store two related pieces of data in a single *document*.

Operations within a document are less expensive for the server than operations that involve multiple documents.

In general, use embedded data models when:

- you have "contains" relationships between entities. See *Model Embedded One-to-One Relationships Between Documents* (page 113).
- you have one-to-many relationships where the "many" objects always appear with or are viewed in the context of their parent documents. See *Model Embedded One-to-Many Relationships Between Documents* (page 114).

Embedding provides the following benefits:

- generally better performance for read operations.
- the ability to request and retrieve related data in a single database operation.

Embedding related data in documents, can lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size` (page 1139). For larger documents, consider using *GridFS* (page 70).

See also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 44) for more examples on accessing arrays.
- *Subdocuments* (page 43) for more examples on accessing subdocuments.

Referencing

To normalize data, store *references* (page 1150) between two documents to indicate a relationship between the data represented in each document.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets. See *data-modeling-trees*.

Referencing provides more flexibility than embedding; however, to resolve the references, client-side applications must issue follow-up queries. In other words, using references requires more roundtrips to the server.

See *Model Referenced One-to-Many Relationships Between Documents* (page 115) for an example of referencing.

Atomicity

MongoDB only provides atomic operations on the level of a single document. [1] As a result needs for atomic operations influence decisions to use embedded or referenced relationships when modeling data for MongoDB.

Embed fields that need to be modified together atomically in the same document. See *Model Data for Atomic Operations* (page 117) for an example of atomic updates within a single document.

8.1.3 Operational Considerations

In addition to normalization and normalization concerns, a number of other operational factors help shape data modeling decisions in MongoDB. These factors include:

- data lifecycle management,
- number of collections and
- indexing requirements,
- sharding, and
- managing document growth.

These factors implications for database and application performance as well as future maintenance and development costs.

Data Lifecycle Management

Data modeling decisions should also take data lifecycle management into consideration.

The *Time to Live or TTL feature* (page 599) of collections expires documents after a period of time. Consider using the TTL feature if your application requires some data to persist in the database for a limited period of time.

Additionally, if your application only uses recently inserted documents consider *Capped Collections* (page 578). Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and optimized to support operations that insert and read documents based on insertion order.

Large Number of Collections

In certain situations, you might choose to store information in several collections rather than in a single collection.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

```
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ...}
```

If the total number of documents is low you may group documents into collection by type. For logs, consider maintaining distinct log collections, such as `logs.dev` and `logs.debug`. The `logs.dev` collection would contain only the documents related to the dev environment.

Generally, having large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.

[1] Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple subdocuments within that single record are still atomic.

- Each index, including the index on `_id`, requires at least 8KB of data space.

A single `<database>.ns` file stores all meta-data for each *database*. Each index and collection has its own entry in the namespace file, MongoDB places `limits on the size of namespace files` (page 1139).

Because of `limits on namespaces` (page 1139), you may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support, as in the following example:

```
db.system.namespaces.count()
```

The `<database>.ns` file defaults to 16 MB. To change the size of the `<database>.ns` file, pass a new size to *`--nssize option <new size MB>`* on server start.

The *`--nssize`* sets the size for *new* `<database>.ns` files. For existing databases, after starting up the server with *`--nssize`*, run the `db.repairDatabase()` (page 1011) command from the `mongo` (page 1066) shell.

Indexes

Create indexes to support common queries. Generally, indexes and index use in MongoDB correspond to indexes and index use in relational database: build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the `_id` field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive as each insert must add keys to each index.
- Collections with high proportion of read operations to write operations often benefit from additional indexes. Indexes do not affect un-indexed read operations.

See *Indexing Strategies* (page 321) for more information on determining indexes. Additionally, the MongoDB *database profiler* (page 175) may help identify inefficient queries.

Sharding

Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` (page 1049) instances or *shards*.

The shard key determines how MongoDB distributes data among shards in a sharded collection. Selecting the proper *shard key* (page 502) has significant implications for performance.

See *Sharding Introduction* (page 487) and *Shard Keys* (page 502) for more information.

Document Growth

Certain updates to documents can increase the document size, such as pushing elements to an array and adding new fields. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. This internal relocation can be both time and resource consuming.

Although MongoDB automatically provides padding to minimize the occurrence of relocations, you may still need to manually handle document growth. Refer to *Pre-Aggregated Reports* (page 641) for an example of the *Pre-allocation* approach to handle document growth.

8.1.4 Data Modeling Patterns and Examples

The following documents provide overviews of various data modeling patterns and common schema design considerations:

- *Model Embedded One-to-One Relationships Between Documents* (page 113)
- *Model Embedded One-to-Many Relationships Between Documents* (page 114)
- *Model Referenced One-to-Many Relationships Between Documents* (page 115)
- *Model Data for Atomic Operations* (page 117)
- *Model Tree Structures with Parent References* (page 118)
- *Model Tree Structures with Child References* (page 118)
- *Model Tree Structures with Materialized Paths* (page 120)
- *Model Tree Structures with Nested Sets* (page 121)

For more information and examples of real-world data modeling, consider the following external resources:

- Schema Design by Example
- Walkthrough MongoDB Data Modeling
- Document Design for MongoDB
- Dynamic Schema Blog Post
- MongoDB Data Modeling and Rails
- Ruby Example of Materialized Paths
- Sean Cribs Blog Post which was the source for much of the *data-modeling-trees* content.

CHAPTER 9

Data Modeling Patterns

9.1 Model Embedded One-to-One Relationships Between Documents

9.1.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 108) documents to describe relationships between connected data.

9.1.2 Pattern

Consider the following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between `patron` and `address` data, the `address` belongs to the `patron`.

In the normalized data model, the `address` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA"
   zip: 12345
}
```

If the `address` data is frequently retrieved with the `name` information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the `address` data in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
```

```
            street: "123 Fake Street",
            city: "Faketon",
            state: "MA"
            zip: 12345
          }
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

9.2 Model Embedded One-to-Many Relationships Between Documents

9.2.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 108) documents to describe relationships between connected data.

9.2.2 Pattern

Consider the following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between `patron` and `address` data, the `patron` has multiple `address` entities.

In the normalized data model, the `address` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: 12345
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: 12345
}
```

If your application frequently retrieves the `address` data with the `name` information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the `address` data entities in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: 12345
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: 12345
                }
              ]
 }
```

With the embedded data model, your application can retrieve the complete patron information with one query.

9.3 Model Referenced One-to-Many Relationships Between Documents

9.3.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 108) between documents to describe relationships between connected data.

9.3.2 Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

```
{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
```

```
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"
}
```

9.4 Model Data for Atomic Operations

9.4.1 Pattern

Consider the following example that keeps a library book and its checkout information. The example illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.

Consider the following `book` document that stores the number of available copies for checkout and the current checkout information:

```
book = {
          _id: 123456789,
          title: "MongoDB: The Definitive Guide",
          author: [ "Kristina Chodorow", "Mike Dirolf" ],
          published_date: ISODate("2010-09-24"),
          pages: 216,
          language: "English",
          publisher_id: "oreilly",
          available: 3,
          checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
        }
```

You can use the `db.collection.findAndModify()` (page 952) method to atomically determine if a book is available for checkout and update with the new checkout information. Embedding the `available` field and the `checkout` field within the same document ensures that the updates to these fields are in sync:

```
db.books.findAndModify ( {
   query: {
            _id: 123456789,
            available: { $gt: 0 }
          },
   update: {
             $inc: { available: -1 },
             $push: { checkout: { by: "abc", date: new Date() } }
           }
} )
```

9.5 Model Tree Structures with Parent References

9.5.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 108) to "parent" nodes in children nodes.

9.5.2 Pattern

The *Parent References* pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node's parent.

Consider the following example that models a tree of categories using *Parent References*:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "Postgres", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).parent
  ```

- You can create an index on the field `parent` to enable fast search by the parent node:

  ```
  db.categories.ensureIndex( { parent: 1 } )
  ```

- You can query by the `parent` field to find its immediate children nodes:

  ```
  db.categories.find( { parent: "Databases" } )
  ```

The *Parent Links* pattern provides a simple solution to tree storage, but requires multiple queries to retrieve subtrees.

9.6 Model Tree Structures with Child References

9.6.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 108) in the parent-nodes to children nodes.

9.6.2 Pattern

The *Child References* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's children.

Consider the following example that models a tree of categories using *Child References*:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "Postgres", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "Postgres" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

- The query to retrieve the immediate children of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "Databases" } ).children
  ```

- You can create an index on the field `children` to enable fast search by the child nodes:

  ```
  db.categories.ensureIndex( { children: 1 } )
  ```

- You can query for a node in the `children` field to find its parent node as well as its siblings:

  ```
  db.categories.find( { children: "MongoDB" } )
  ```

The *Child References* pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.

9.7 Model Tree Structures with an Array of Ancestors

9.7.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using *references* (page 108) to parent nodes and an array that stores all ancestors.

9.7.2 Pattern

The *Array of Ancestors* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's ancestors or path.

Consider the following example that models a tree of categories using *Array of Ancestors*:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "
db.categories.insert( { _id: "Postgres", ancestors: [ "Books", "Programming", "Databases" ], parent:
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programmin
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programmin
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).ancestors
  ```

- You can create an index on the field `ancestors` to enable fast search by the ancestors nodes:

```
db.categories.ensureIndex( { ancestors: 1 } )
```

- You can query by the `ancestors` to find all its descendants:

```
db.categories.find( { ancestors: "Programming" } )
```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the ancestors field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* pattern but is more straightforward to use.

9.8 Model Tree Structures with Materialized Paths

9.8.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

9.8.2 Pattern

The *Materialized Paths* pattern stores each tree node in a document; in addition to the tree node, document stores as a string the id(s) of the node's ancestors or path. Although the *Materialized Paths* pattern requires additional steps of working with strings and regular expressions, the pattern also provides more flexibility in working with the path, such as finding nodes by partial paths.

Consider the following example that models a tree of categories using *Materialized Paths* ; the path string uses the comma `,` as a delimiter:

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "Postgres", path: ",Books,Programming,Databases," } )
```

- You can query to retrieve the whole tree, sorting by the `path`:

```
db.categories.find().sort( { path: 1 } )
```

- You can use regular expressions on the `path` field to find the descendants of `Programming`:

```
db.categories.find( { path: /,Programming,/ } )
```

- You can also retrieve the descendants of `Books` where the `Books` is also at the topmost level of the hierarchy:

```
db.categories.find( { path: /^,Books,/ } )
```

- To create an index on the field `path` use the following invocation:

```
db.categories.ensureIndex( { path: 1 } )
```

This index may improve performance, depending on the query:

- For queries of the `Books` sub-tree (e.g. `http://docs.mongodb.org/manual^,Books,/`) an index on the `path` field improves the query performance significantly.
- For queries of the `Programming` sub-tree (e.g. `http://docs.mongodb.org/manual,Programming,/`), or similar queries of sub-tress, where the node might be in the middle of the indexed string, the query must inspect the entire index.

  For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

9.9 Model Tree Structures with Nested Sets

9.9.1 Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Considerations for MongoDB Applications* (page 107) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree like structure that optimizes discovering subtrees at the expense of tree mutability.

9.9.2 Pattern

The *Nested Sets* pattern identifies each node in the tree as stops in a round-trip traversal of the tree. The application visits each node in the tree twice; first during the initial trip, and second during the return trip. The *Nested Sets* pattern stores each tree node in a document; in addition to the tree node, document stores the id of node's parent, the node's initial stop in the `left` field, and its return stop in the `right` field.

Consider the following example that models a tree of categories using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "Postgres", parent: "Databases", left: 8, right: 9 } )
```

You can query to retrieve the descendants of a node:

```
var databaseCategory = db.v.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } }
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

9.10 Model Data to Support Keyword Search

Note: Keyword search is *not* the same as text search or full text search, and does not provide stemming or other text-processing features. See the *Limitations of Keyword Indexes* (page 122) section for more information.

In 2.4, MongoDB provides a text search feature. See *Text Search* (page 353) for more information.

If your application needs to perform queries on the content of a field that holds text you can perform exact matches on the text or use `$regex` (page 798) to use regular expression pattern matches. However, for many operations on text, these methods do not satisfy application requirements.

This pattern describes one method for supporting keyword search using MongoDB to support application search functionality, that uses keywords stored in an array in the same document as the text field. Combined with a *multi-key index* (page 312), this pattern can support application's keyword search operations.

9.10.1 Pattern

To add structures to your document to support keyword-based queries, create an array field in your documents and add the keywords as strings in the array. You can then create a *multi-key index* (page 312) on the array and create queries that select values from the array.

Example

Given a collection of library volumes that you want to provide topic-based search. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the `Moby-Dick` volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

```
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

9.10.2 Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and *multi-key indexes* (page 312); however, these keyword indexes are not sufficient or comparable to full-text products in the following respects:

- *Stemming*. Keyword queries in MongoDB can not parse keywords for root or related words.
- *Synonyms*. Keyword-based search features must provide support for synonym or related queries in the application layer.
- *Ranking*. The keyword look ups described in this document do not provide a way to weight results.

- *Asynchronous Indexing*. MongoDB builds indexes synchronously, which means that the indexes used for keyword indexes are always current and can operate in real-time. However, asynchronous bulk indexes may be more efficient for some kinds of content and workloads.

Part IV

Administration

The documentation in this section outlines core administrative tasks and practices that operators of MongoDB will want to consider.

CHAPTER 10

Background

10.1 Run-time Database Configuration

The *command line* (page 1049) and *configuration file* (page 1115) interfaces provide MongoDB administrators with a large number of options and settings for controlling the operation of the database system. This document provides an overview of common configurations and examples of best-practice configurations for common use cases.

While both interfaces provide access to the same collection of options and settings, this document primarily uses the configuration file interface. If you run MongoDB using a control script or installed from a package for your operating system, you likely already have a configuration file located at `/etc/mongodb.conf`. Confirm this by checking the content of the `/etc/init.d/mongod` or `/etc/rc.d/mongod` script to insure that the *control scripts* start the `mongod` (page 1049) 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 `/etc/mongodb.conf` file on your system to control the configuration of your database instance.

10.1.1 Configure the Database

Consider the following basic configuration:

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

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

- `fork` (page 1117) is `true`, which enables a *daemon* mode for `mongod` (page 1049), 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 1116) is `127.0.0.1`, which forces the server to only listen for requests on the localhost IP. Only bind to secure interfaces that the application-level systems can access with access control provided by system network filtering (i.e. "*firewall*").

- port (page 1116) is 27017, which is the default MongoDB port for database instances. MongoDB can bind to any port. You can also filter access based on port using network filtering tools.

  Note: UNIX-like systems require superuser privileges to attach processes to ports lower than 1024.

- quiet (page 1123) 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 (page 894) to modify this setting during run time.
- dbpath (page 1118) is http://docs.mongodb.org/manualsrv/mongodb, which specifies where MongoDB will store its data files. http://docs.mongodb.org/manualsrv/mongodb and http://docs.mongodb.org/manualvar/lib/mongodb are popular locations. The user account that mongod (page 1049) runs under will need read and write access to this directory.
- logpath (page 1116) is http://docs.mongodb.org/manualvar/log/mongodb/mongod.log which is where mongod (page 1049) will write its output. If you do not set this value, mongod (page 1049) writes all output to standard output (e.g. stdout.)
- logappend (page 1117) is true, which ensures that mongod (page 1049) does not overwrite an existing log file following the server start operation.
- journal (page 1119) is true, which enables *journaling*. Journaling ensures single instance write-durability. 64-bit builds of mongod (page 1049) 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.

10.1.2 Security Considerations

The following collection of configuration options are useful for limiting access to a mongod (page 1049) 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 1116)" 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 1117)" to true disables the UNIX Socket, which is otherwise enabled by default. This limits access on the local system. This is desirable when running MongoDB on systems with shared access, but in most situations has minimal impact.
- "auth (page 1118)" is true enables the authentication system within MongoDB. If enabled you will need to log in by connecting over the localhost interface for the first time to create user credentials.

See also:

Security Practices and Management (page 207)

10.1.3 Replication and Sharding Configuration

Replication Configuration

Replica set configuration is straightforward, and only requires that the `replSet` (page 1124) 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` (page 1066) shell to add hosts to the replica set.

See also:

Replica set reconfiguration (page 477).

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

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

New in version 1.8: for replica sets, and 1.9.1 for sharded replica sets.

Setting `keyFile` (page 1117) 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 1061) instances that connect to the set. The keyfile must be less than one kilobyte in size and may only contain characters in the base64 set and the file must not have group or "world" permissions on UNIX systems.

See also:

The "*Replica set Reconfiguration* (page 477)" section for information regarding the process for changing replica set during operation.

Additionally, consider the "*Replica Set Security* (page 212)" section for information on configuring authentication with replica sets.

Finally, see the "*Replication* (page 367)" document for more information on replication in MongoDB and replica set configuration in general.

Sharding Configuration

Sharding requires a number of `mongod` (page 1049) 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 129) `mongod` (page 1049) 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 1061)" and "`mongod` (page 1049)" instances.

To set up shards, configure two or more `mongod` (page 1049) instance using your *base configuration* (page 129), adding the `shardsvr` (page 1125) setting:

```
shardsvr = true
```

Finally, to establish the cluster, configure at least one mongos (page 1061) process with the following settings:

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

You can specify multiple configdb (page 1126) instances by specifying hostnames and ports in the form of a comma separated list. In general, avoid modifying the chunkSize (page 1126) from the default value of 64, [1] and *should* ensure this setting is consistent among all mongos (page 1061) instances.

See also:

The "*Sharding* (page 485)" section of the manual for more information on sharding and cluster configuration.

10.1.4 Run Multiple Database Instances on the Same System

In many cases running multiple instances of mongod (page 1049) 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 (page 1049) on a single system.

In these cases, use a *base configuration* (page 129) for each instance, but consider the following configuration values:

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

The dbpath (page 1118) value controls the location of the mongod (page 1049) instance's data directory. Ensure that each database has a distinct and well labeled data directory. The pidfilepath (page 1117) controls where mongod (page 1049) process places it's *process id* file. As this tracks the specific mongod (page 1049) 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.

10.1.5 Diagnostic Configurations

The following configuration options control various mongod (page 1049) 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 129) and add these options if you are experiencing some unknown issue or performance problem as needed:

- slowms (page 1121) configures the threshold for the *database profiler* to consider a query "slow." The default value is 100 milliseconds. Set a lower value if the database profiler does not return useful results. See *Optimization Strategies for MongoDB* (page 576) 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 (page 1049) instances. Additionally, you may find that multiple databases with small working sets may function acceptably on a single system.

- `profile` (page 1121) 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 1115) enables a verbose logging mode that modifies `mongod` (page 1049) 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 1118) enables *diagnostic logging*. Level `3` logs all read and write options.
- `objcheck` (page 1116) forces `mongod` (page 1049) 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 1118) forces `mongod` (page 1049) 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.

10.2 Backup and Recovery Operations for MongoDB

10.2.1 Backup Strategies for MongoDB Systems

Backups are an important part of any operational disaster recovery plan. A good backup plan must be able to capture data in a consistent and usable state, and operators must be able to automate both the backup and the recovery operations. Also test all components of the backup system to ensure that you can recover backed up data as needed. If you cannot effectively restore your database from the backup, then your backups are useless. This document addresses higher level backup strategies, consider the *Backup and Recovery Operations for MongoDB* (page 133) tutorials for documentation of specific backup procedures.

Note: 10gen offers the MongoDB Management Service that supports backup and restoration for MongoDB deployments. See the MMS Backup Documentation for more information.

Backup Considerations

As you develop a backup strategy for your MongoDB deployment consider the following factors:

- Geography. Ensure that you move some backups away from the your primary database infrastructure.
- System errors. Ensure that your backups can survive situations where hardware failures or disk errors impact the integrity or availability of your backups.
- Production constraints. Backup operations themselves sometimes require substantial system resources. It is important to consider the time of the backup schedule relative to peak usage and maintenance windows.
- System capabilities. Some of the block-level snapshot tools require special support on the operating-system or infrastructure level.

- Database configuration. *Replication* and *sharding* can affect the process and impact of the backup implementation. See *Sharded Cluster Backup Considerations* (page 134) and *Replica Set Backup Considerations* (page 135).
- 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.

Approaches to Backing Up MongoDB Systems

There are two main methodologies for backing up MongoDB instances. Creating binary "dumps" of the database using `mongodump` (page 1075) or creating filesystem level snapshots. Both methodologies have advantages and disadvantages:

- binary database dumps are comparatively small, because they don't include index content or pre-allocated free space, and *record padding* (page 55). However, it's impossible to capture a copy of a running system that reflects a single moment in time using a binary dump.
- filesystem snapshots, sometimes called block level backups, produce larger backup sizes, but complete quickly and can reflect a single moment in time on a running system. However, snapshot systems require filesystem and operating system support and tools.

The best option depends on the requirements of your deployment and disaster recovery needs. Typically, filesystem snapshots are because of their accuracy and simplicity; however, `mongodump` (page 1075) is a viable option used often to generate backups of MongoDB systems.

The following documents provide details and procedures on the two approaches:

- *Use Filesystem Snapshots to Backup and Restore MongoDB Databases* (page 138).
- *Use mongodump and mongorestore to Backup and Restore MongoDB Databases* (page 135).

In some cases, taking backups is difficult or impossible because of large data volumes, distributed architectures, and data transmission speeds. In these situations, increase the number of members in your replica set or sets.

Backup Strategies for MongoDB Deployments

Sharded Cluster Backup Considerations

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Sharded clusters complicate backup operations, as distributed systems. True point-in-time backups are only possible when stopping all write activity from the application. To create a precise moment-in-time snapshot of a cluster, stop all application write activity to the database, capture a backup, and allow only write operations to the database after the backup is complete.

However, you can capture a backup of a cluster that **approximates** a point-in-time backup by capturing a backup from a secondary member of the replica sets that provide the shards in the cluster at roughly the same moment. If you decide to use an approximate-point-in-time backup method, ensure that your application can operate using a copy of the data that does not reflect a single moment in time.

The following documents describe sharded cluster related backup procedures:

- *Backup a Small Sharded Cluster with mongodump* (page 146)
- *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 147)
- *Create Backup of a Sharded Cluster with Database Dumps* (page 148)

- *Schedule Backup Window for Sharded Clusters* (page 151)
- *Restore a Single Shard* (page 149)
- *Restore Sharded Clusters* (page 150)

Replica Set Backup Considerations

In most cases, backing up data stored in a *replica set* is similar to backing up data stored in a single instance. Options include:

- Create a file system snapshot of a single *secondary*, as described in *Use mongodump and mongorestore to Backup and Restore MongoDB Databases* (page 135). You may choose to maintain a dedicated *hidden member* for backup purposes.
- As an alternative you can create a backup with the `mongodump` (page 1075) program and the `--oplog` option. To restore this backup use the `mongorestore` (page 1079) program and the `--oplogReplay` option.

If you have a *sharded cluster* where each *shard* is itself a replica set, you can use one of these methods to create a backup of the entire cluster without disrupting the operation of the node. In these situations you should still turn off the balancer when you create backups.

For any cluster, using a non-primary node to create backups is particularly advantageous in that the backup operation does not affect the performance of the primary. Replication itself provides some measure of redundancy. Nevertheless, keeping point-in time backups of your cluster to provide for disaster recovery and as an additional layer of protection is crucial.

For an overview of backup strategies and considerations for all MongoDB deployments, consider, *Backup Strategies for MongoDB Systems* (page 133).

Note: 10gen offers the MongoDB Management Service that supports backup and restoration for MongoDB deployments. See the MMS Backup Documentation for more information.

For practical instructions and example backup procedures consider the documents in this section.

10.2.2 Backup and Recovery Procedures

Use `mongodump` and `mongorestore` to Backup and Restore MongoDB Databases

This document describes the process for writing the entire contents of your MongoDB instance to a file in a binary format. If disk-level snapshots are not available, this approach provides the best option for full system database backups. If your system has disk level snapshot capabilities, consider the backup methods described in *Use Filesystem Snapshots to Backup and Restore MongoDB Databases* (page 138).

See also:

Backup Strategies for MongoDB Systems (page 133), *mongodump* (page 1075), and *mongorestore* (page 1079).

Backup a Database with `mongodump`

Important: `mongodump` (page 1075) does *not* create output for the `local` database.

Basic mongodump Operations The mongodump (page 1075) utility can back up data by either:

- connecting to a running mongod (page 1049) or mongos (page 1061) instance, or
- accessing data files without an active instance.

The utility can create a backup for an entire server, database or collection, or can use a query to backup just part of a collection.

When you run mongodump (page 1075) without any arguments, the command connects to the MongoDB instance on the local system (e.g. `127.0.0.1` or `localhost`) on port `27017` and creates a database backup named `dump/` in the current directory.

To backup data from a mongod (page 1049) or mongos (page 1061) instance running on the same machine and on the default port of `27017` use the following command:

```
mongodump
```

Note: The format of data created by mongodump (page 1075) tool from the 2.2 distribution or later is different and incompatible with earlier versions of mongod (page 1049).

To limit the amount of data included in the database dump, you can specify *--db* and *--collection* as options to the mongodump (page 1075) command. For example:

```
mongodump --dbpath /data/db/ --out /data/backup/
```

```
mongodump --host mongodb.example.net --port 27017
```

mongodump (page 1075) will write *BSON* files that hold a copy of data accessible via the mongod (page 1049) listening on port `27017` of the `mongodb.example.net` host.

```
mongodump --collection collection --db test
```

This command creates a dump of the collection named `collection` from the database `test` in a `dump/` subdirectory of the current working directory.

Point in Time Operation Using Oplogs Use the *--oplog* option with mongodump (page 1075) to collect the *oplog* entries to build a point-in-time snapshot of a database within a replica set. With *--oplog*, mongodump (page 1075) copies all the data from the source database as well as all of the *oplog* entries from the beginning of the backup procedure to until the backup procedure completes. This backup procedure, in conjunction with *mongorestore --oplogReplay*, allows you to restore a backup that reflects a consistent and specific moment in time.

Create Backups Without a Running mongod Instance If your MongoDB instance is not running, you can use the *--dbpath* option to specify the location to your MongoDB instance's database files. mongodump (page 1075) reads from the data files directly with this operation. This locks the data directory to prevent conflicting writes. The mongod (page 1049) process must *not* be running or attached to these data files when you run mongodump (page 1075) in this configuration. Consider the following example:

```
mongodump --dbpath /srv/mongodb
```

Create Backups from Non-Local mongod Instances The *--host* and *--port* options for mongodump (page 1075) allow you to connect to and backup from a remote host. Consider the following example:

```
mongodump --host mongodb1.example.net --port 3017 --username user --password pass --out /opt/backup/r
```

On any mongodump (page 1075) command you may, as above, specify username and password credentials to specify database authentication.

Restore a Database with `mongorestore`

The mongorestore (page 1079) utility restores a binary backup created by mongodump (page 1075). By default, mongorestore (page 1079) looks for a database backup in the `dump/` directory.

The mongorestore (page 1079) utility can restore data either by:

- connecting to a running mongod (page 1049) or mongos (page 1061) directly, or
- writing to a set of MongoDB data files without use of a running mongod (page 1049).

The mongorestore (page 1079) utility can restore either an entire database backup or a subset of the backup.

A mongorestore (page 1079) command that connects to an active mongod (page 1049) or mongos (page 1061) has the following prototype form:

```
mongorestore --port <port number> <path to the backup>
```

A mongorestore (page 1079) command that writes to data files without using a running mongod (page 1049) has the following prototype form:

```
mongorestore --dbpath <database path> <path to the backup>
```

Consider the following example:

```
mongorestore dump-2012-10-25/
```

Here, mongorestore (page 1079) imports the database backup in the `dump-2012-10-25` directory to the mongod (page 1049) instance running on the localhost interface.

Restore Point in Time Oplog Backup If you created your database dump using the *--oplog* option to ensure a point-in-time snapshot, call mongorestore (page 1079) with the *--oplogReplay* option, as in the following example:

```
mongorestore --oplogReplay
```

You may also consider using the *mongorestore --objcheck* option to check the integrity of objects while inserting them into the database, or you may consider the *mongorestore --drop* option to drop each collection from the database before restoring from backups.

Restore a Subset of data from a Binary Database Dump mongorestore (page 1079) 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 (page 1079) only adds documents to the database from the dump located in the `dump/` folder *if* the documents have a field name `field` that holds a value of `1`. Enclose the filter in single quotes (e.g. `'`) to prevent the filter from interacting with your shell environment.

Restore without a Running `mongod` mongorestore (page 1079) can write data to MongoDB data files without needing to connect to a mongod (page 1049) directly.

```
mongorestore --dbpath /srv/mongodb --journal
```

Here, `mongorestore` (page 1079) restores the database dump located in `dump/` folder into the data files located at `http://docs.mongodb.org/manualsrv/mongodb`. Additionally, the `--journal` option ensures that `mongorestore` (page 1079) 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 1075) and *mongorestore* (page 1079).

Restore Backups to Non-Local mongod Instances By default, `mongorestore` (page 1079) connects to a MongoDB instance running on the localhost interface (e.g. `127.0.0.1`) and on the default port (`27017`). If you want to restore to a different host or port, use the `--host` and `--port` options.

Consider the following example:

```
mongorestore --host mongodb1.example.net --port 3017 --username user --password pass /opt/backup/mong
```

As above, you may specify username and password connections if your `mongod` (page 1049) requires authentication.

Use Filesystem Snapshots to Backup and Restore MongoDB Databases

This document describes a procedure for creating backups of MongoDB systems using system-level tools, such as *LVM* or storage appliance, as well as the corresponding restoration strategies.

These filesystem snapshots, or "block-level" backup methods use system level tools to create copies of the device that holds MongoDB's data files. These methods complete quickly and work reliably, but require more system configuration outside of MongoDB.

See also:

Backup Strategies for MongoDB Systems (page 133) and *Use mongodump and mongorestore to Backup and Restore MongoDB Databases* (page 135).

Snapshots Overview

Snapshots work by creating pointers between the live data and a special snapshot volume. These pointers are theoretically equivalent to "hard links." As the working data diverges from the snapshot, the snapshot process uses a copy-on-write strategy. As a result the snapshot only stores modified data.

After making the snapshot, you mount the snapshot image on your file system and copy data from the snapshot. The resulting backup contains a full copy of all data.

Snapshots have the following limitations:

- The database must be in a consistent or recoverable state when the snapshot takes place. This means that all writes accepted by the database need to be fully written to disk: either to the *journal* or to data files.

  If all writes are not on disk when the backup occurs, the backup will not reflect these changes. If writes are *in progress* when the backup occurs, the data files will reflect an inconsistent state. With *journaling* all data-file states resulting from in-progress writes are recoverable; without journaling you must flush all pending writes to disk before running the backup operation and must ensure that no writes occur during the entire backup procedure.

  If you do use journaling, the journal **must** reside on the same volume as the data.

- Snapshots create an image of an entire disk image. Unless you need to back up your entire system, consider isolating your MongoDB data files, journal (if applicable), and configuration on one logical disk that doesn't contain any other data.

Alternately, store all MongoDB data files on a dedicated device so that you can make backups without duplicating extraneous data.

- Ensure that you copy data from snapshots and onto other systems to ensure that data is safe from site failures.
- Although different snapshots methods provide different capability, the LVM method outlined below does not provide any capacity for capturing incremental backups.

Snapshots With Journaling If your `mongod` (page 1049) instance has journaling enabled, then you can use any kind of file system or volume/block level snapshot tool to create backups.

If you manage your own infrastructure on a Linux-based system, configure your system with *LVM* to provide your disk packages and provide snapshot capability. You can also use LVM-based setups *within* a cloud/virtualized environment.

Note: Running *LVM* provides additional flexibility and enables the possibility of using snapshots to back up MongoDB.

Snapshots with Amazon EBS in a RAID 10 Configuration If your deployment depends on Amazon's Elastic Block Storage (EBS) with RAID configured within your instance, it is impossible to get a consistent state across all disks using the platform's snapshot tool. As an alternative, you can do one of the following:

- Flush all writes to disk and create a write lock to ensure consistent state during the backup process.

  If you choose this option see *Create Backups on Instances that do not have Journaling Enabled* (page 141).
- Configure *LVM* to run and hold your MongoDB data files on top of the RAID within your system.

  If you choose this option, perform the LVM backup operation described in *Create a Snapshot* (page 139).

Backup and Restore Using LVM on a Linux System

This section provides an overview of a simple backup process using *LVM* on a Linux system. While the tools, commands, and paths may be (slightly) different on your system the following steps provide a high level overview of the backup operation.

Note: Only use the following procedure as a guideline for a backup system and infrastructure. Production backup systems must consider a number of application specific requirements and factors unique to specific environments.

Create a Snapshot To create a snapshot with *LVM*, issue a command as root in the following format:

```
lvcreate --size 100M --snapshot --name mdb-snap01 /dev/vg0/mongodb
```

This command creates an *LVM* snapshot (with the `--snapshot` option) named `mdb-snap01` of the `mongodb` volume in the `vg0` volume group.

This example creates a snapshot named `mdb-snap01` located at `http://docs.mongodb.org/manualdev/vg0/mdb-snap01` The location and paths to your systems volume groups and devices may vary slightly depending on your operating system's *LVM* configuration.

The snapshot has a cap of at 100 megabytes, because of the parameter `--size 100M`. This size does not reflect the total amount of the data on the disk, but rather the quantity of differences between the current state of `http://docs.mongodb.org/manualdev/vg0/mongodb` and the creation of the snapshot (i.e. `http://docs.mongodb.org/manualdev/vg0/mdb-snap01`.)

> **Warning:** Ensure that you create snapshots with enough space to account for data growth, particularly for the period of time that it takes to copy data out of the system or to a temporary image.
> If your snapshot runs out of space, the snapshot image becomes unusable. Discard this logical volume and create another.

The snapshot will exist when the command returns. You can restore directly from the snapshot at any time or by creating a new logical volume and restoring from this snapshot to the alternate image.

While snapshots are great for creating high quality backups very quickly, they are not ideal as a format for storing backup data. Snapshots typically depend and reside on the same storage infrastructure as the original disk images. Therefore, it's crucial that you archive these snapshots and store them elsewhere.

Archive a Snapshot After creating a snapshot, mount the snapshot and move the data to separate storage. Your system might try to compress the backup images as you move the offline. The following procedure fully archives the data from the snapshot:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | gzip > mdb-snap01.gz
```

The above command sequence does the following:

- Ensures that the `http://docs.mongodb.org/manualdev/vg0/mdb-snap01` device is not mounted.
- Performs a block level copy of the entire snapshot image using the `dd` command and compresses the result in a gzipped file in the current working directory.

  > **Warning:** This command will create a large `gz` file in your current working directory. Make sure that you run this command in a file system that has enough free space.

Restore a Snapshot To restore a snapshot created with the above method, issue the following sequence of commands:

```
lvcreate --size 1G --name mdb-new vg0
gzip -d -c mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

The above sequence does the following:

- Creates a new logical volume named `mdb-new`, in the `http://docs.mongodb.org/manualdev/vg0` volume group. The path to the new device will be `http://docs.mongodb.org/manualdev/vg0/mdb-new`.

  > **Warning:** This volume will have a maximum size of 1 gigabyte. The original file system must have had a total size of 1 gigabyte or smaller, or else the restoration will fail.
  > Change `1G` to your desired volume size.

- Uncompresses and unarchives the `mdb-snap01.gz` into the `mdb-new` disk image.
- Mounts the `mdb-new` disk image to the `http://docs.mongodb.org/manualsrv/mongodb` directory. Modify the mount point to correspond to your MongoDB data file location, or other location as needed.

Note: The restored snapshot will have a stale `mongod.lock` file. If you do not remove this file from the snapshot, and MongoDB may assume that the stale lock file indicates an unclean shutdown. If you're running with `journal` (page 1119) enabled, and you *do not* use `db.fsyncLock()` (page 1004), you do not need to remove the `mongod.lock` file. If you use `db.fsyncLock()` (page 1004) you will need to remove the lock.

Restore Directly from a Snapshot To restore a backup without writing to a compressed gz file, use the following sequence of commands:

```
umount /dev/vg0/mdb-snap01
lvcreate --size 1G --name mdb-new vg0
dd if=/dev/vg0/mdb-snap01 of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Remote Backup Storage You can implement off-system backups using the *combined process* (page 141) and SSH.

This sequence is identical to procedures explained above, except that it archives and compresses the backup on a remote system using SSH.

Consider the following procedure:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | ssh username@example.com gzip > /opt/backup/mdb-snap01.gz
lvcreate --size 1G --name mdb-new vg0
ssh username@example.com gzip -d -c /opt/backup/mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Create Backups on Instances that do not have Journaling Enabled

If your `mongod` (page 1049) instance does not run with journaling enabled, or if your journal is on a separate volume, obtaining a functional backup of a consistent state is more complicated. As described in this section, you must flush all writes to disk and lock the database to prevent writes during the backup process. If you have a *replica set* configuration, then for your backup use a *secondary* which is not receiving reads (i.e. *hidden member*).

1. To flush writes to disk and to "lock" the database (to prevent further writes), issue the `db.fsyncLock()` (page 1004) method in the `mongo` (page 1066) shell:

   ```
   db.fsyncLock();
   ```

2. Perform the backup operation described in *Create a Snapshot* (page 139).

3. To unlock the database after the snapshot has completed, use the following command in the `mongo` (page 1066) shell:

   ```
   db.fsyncUnlock();
   ```

   Note: Changed in version 2.0: MongoDB 2.0 added `db.fsyncLock()` (page 1004) and `db.fsyncUnlock()` (page 1004) helpers to the `mongo` (page 1066) shell. Prior to this version, use the `fsync` (page 888) 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()` (page 1004) while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()` (page 1004). Disable profiling using `db.setProfilingLevel()` (page 1012) as follows in the `mongo` (page 1066) shell:

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

> **Warning:** Changed in version 2.2: When used in combination with `fsync` (page 888) or `db.fsyncLock()` (page 1004), `mongod` (page 1049) may block some reads, including those from `mongodump` (page 1075), when queued write operation waits behind the `fsync` (page 888) lock.

Copy Databases Between Instances

Synopsis

MongoDB provides the `copydb` (page 883) and `clone` (page 885) *database commands* to support migrations of entire logical databases between `mongod` (page 1049) instances. With these commands you can copy data between instances with a simple interface without the need for an intermediate stage. The `db.cloneDatabase()` (page 996) and `db.copyDatabase()` (page 996) provide helpers for these operations in the `mongo` (page 1066) shell.

Data migrations that require an intermediate stage or that involve more than one database instance are beyond the scope of this tutorial. `copydb` (page 883) and `clone` (page 885) are more ideal for use cases that resemble the following use cases:

- data migrations,
- data warehousing, and
- seeding test environments.

Also consider the *Backup Strategies for MongoDB Systems* (page 133) and *Import and Export MongoDB Data* (page 166) documentation for more related information.

Note: `copydb` (page 883) and `clone` (page 885) do not produce point-in-time snapshots of the source database. Write traffic to the source or destination database during the copy process will result divergent data sets.

Considerations

- You must run `copydb` (page 883) or `clone` (page 885) on the destination server.
- You cannot use `copydb` (page 883) or `clone` (page 885) with databases that have a sharded collection in a *sharded cluster*, or any database via a `mongos` (page 1061).
- You *can* use `copydb` (page 883) or `clone` (page 885) with databases that do not have sharded collections in a *cluster* when you're connected directly to the `mongod` (page 1049) instance.
- You can run `copydb` (page 883) or `clone` (page 885) commands on a *secondary* member of a replica set, with properly configured *read preference*.
- Each destination `mongod` (page 1049) instance must have enough free disk space on the destination server for the database you are copying. Use the `db.stats()` (page 1013) operation to check the size of the database on the source `mongod` (page 1049) instance. For more information, see `db.stats()` (page 1013).

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` (page 883) command, or the `db.copyDatabase()` (page 996) helper in the `mongo` (page 1066) 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 (page 1049) instance running on the db0.example.net host.
- Connect to the destination server, running on the db1.example.net host, using the mongo (page 1066) shell.
- Model your operation on the following command:

```
db.copyDatabase( "test", "records", "db0.example.net" )
```

Rename a Database You can also use copydb (page 883) or the db.copyDatabase() (page 996) 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 (page 1049) instance:

- Connect to the mongod (page 1049) using the mongo (page 1066) 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 (page 883) command or the db.copyDatabase() (page 996) helper in the mongo (page 1066) shell.

In the following operation, you will copy the test database from the mongod (page 1049) running on db0.example.net to the records database on the local instance (e.g. db1.example.net.) Because the mongod (page 1049) instance running on db0.example.net requires authentication for all connections, you will need to pass db.copyDatabase() (page 996) authentication credentials, as in the following procedure:

- Connect to the destination mongod (page 1049) instance running on the db1.example.net host using the mongo (page 1066) 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 (page 885) command copies a database between mongod (page 1049) instances like copydb (page 883); however, clone (page 885) preserves the database name from the source instance on the destination mongod (page 1049).

For many operations, clone (page 885) is functionally equivalent to copydb (page 883), but it has a more simple syntax and a more narrow use. The mongo (page 1066) shell provides the db.cloneDatabase() (page 996) helper as a wrapper around clone (page 885).

You can use the following procedure to clone a database from the mongod (page 1049) instance running on db0.example.net to the mongod (page 1049) running on db1.example.net:

- Connect to the destination mongod (page 1049) instance running on the db1.example.net host using the mongo (page 1066) 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 (page 885) operation:

```
db.cloneDatabase( "db0.example.net" )
```

Recover MongoDB Data following Unexpected Shutdown

If MongoDB does not shutdown cleanly [3] the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption. [4]

To prevent data inconsistency and corruption, always shut down the database cleanly and use the *durability journaling* (page 1119). MongoDB writes data to the journal, by default, every 100 milliseconds, such that MongoDB can always recover to a consistent state even in the case of an unclean shutdown due to power loss or other system failure.

If you are *not* running as part of a *replica set* **and** do *not* have journaling enabled, use the following procedure to recover data that may be in an inconsistent state. If you are running as part of a replica set, you should *always* restore from a backup or restart the `mongod` (page 1049) instance with an empty `dbpath` (page 1118) and allow MongoDB to perform an initial sync to restore the data.

See also:

The *Administration* (page 127) documents, including *Replica Set Syncing* (page 405), and the documentation on the `repair` (page 1121), `repairpath` (page 1121), and `journal` (page 1119) settings.

Process

Indications When you are aware of a `mongod` (page 1049) instance running without journaling that stops unexpectedly **and** you're not running with replication, you should always run the repair operation before starting MongoDB again. If you're using replication, then restore from a backup and allow replication to perform an initial *sync* (page 405) to restore data.

If the `mongod.lock` file in the data directory specified by `dbpath` (page 1118), `http://docs.mongodb.org/manualdata/db` by default, is *not* a zero-byte file, then `mongod` (page 1049) will refuse to start, and you will find a message that contains the following line in your MongoDB log our output:

```
Unclean shutdown detected.
```

This indicates that you need to run `mongod` (page 1049) with the *--repair* option. If you run repair when the `mongodb.lock` file exists in your `dbpath` (page 1118), or the optional *--repairpath*, you will see a message that contains the following line:

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

If you see this message, as a last resort you may remove the lockfile **and** run the repair operation before starting the database normally, as in the following procedure:

Overview

> **Warning:** Recovering a member of a replica set.
> Do not use this procedure to recover a member of a *replica set*. Instead you should either restore from a *backup* (page 133) or perform an initial sync using data from an intact member of the set, as described in *Resync a Member of a Replica Set* (page 443).

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

[3] To ensure a clean shut down, use the *mongod --shutdown* option, your control script, "Control-C" (when running `mongod` (page 1049) in interactive mode,) or `kill $(pidof mongod)` or `kill -2 $(pidof mongod)`.

[4] You can also use the `db.collection.validate()` (page 976) method to test the integrity of a single collection. However, this process is time consuming, and without journaling you can safely assume that the data is in an invalid state and you should either run the repair operation or resync from an intact member of the replica set.

1. Use the `--repair` option in conjunction with the `--repairpath` option. `mongod` (page 1049) will read the existing data files, and write the existing data to new data files. This does not modify or alter the existing data files.

   You do not need to remove the `mongod.lock` file before using this procedure.

2. Use the `--repair` option. `mongod` (page 1049) will read the existing data files, write the existing data to new files and replace the existing, possibly corrupt, files with new files.

   You must remove the `mongod.lock` file before using this procedure.

Note: `--repair` functionality is also available in the shell with the `db.repairDatabase()` (page 1011) helper for the `repairDatabase` (page 895) command.

Procedures To repair your data files using the `--repairpath` option to preserve the original data files unmodified:

1. Start `mongod` (page 1049) using `--repair` to read the existing data files.

   ```
   mongod --dbpath /data/db --repair --repairpath /data/db0
   ```

   When this completes, the new repaired data files will be in the `http://docs.mongodb.org/manualdata/db0` directory.

2. Start `mongod` (page 1049) using the following invocation to point the `dbpath` (page 1118) at `http://docs.mongodb.org/manualdata/db0`:

   ```
   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/manualdata/db` directory.

To repair your data files without preserving the original files, do not use the `--repairpath` option, as in the following procedure:

1. Remove the stale lock file:

   ```
   rm /data/db/mongod.lock
   ```

   Replace `http://docs.mongodb.org/manualdata/db` with your `dbpath` (page 1118) where your MongoDB instance's data files reside.

   Warning: After you remove the `mongod.lock` file you *must* run the `--repair` process before using your database.

2. Start `mongod` (page 1049) using `--repair` to read the existing data files.

   ```
   mongod --dbpath /data/db --repair
   ```

   When this completes, the repaired data files will replace the original data files in the `http://docs.mongodb.org/manualdata/db` directory.

3. Start `mongod` (page 1049) using the following invocation to point the `dbpath` (page 1118) at `http://docs.mongodb.org/manualdata/db`:

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

mongod.lock

In normal operation, you should **never** remove the `mongod.lock` file and start `mongod` (page 1049). Instead consider the one of the above methods to recover the database and remove the lock files. In dire situations you can remove the lockfile, and start the database using the possibly corrupt files, and attempt to recover data from the database; however, it's impossible to predict the state of the database in these situations.

If you are not running with journaling, and your database shuts down unexpectedly for *any* reason, you should always proceed *as if* your database is in an inconsistent and likely corrupt state. If at all possible restore from *backup* (page 133) or, if running as a *replica set*, restore by performing an initial sync using data from an intact member of the set, as described in *Resync a Member of a Replica Set* (page 443).

10.2.3 Backup and Restore Sharded Clusters

Backup a Small Sharded Cluster with `mongodump`

Overview

If your *sharded cluster* holds a small data set, you can connect to a `mongos` (page 1061) using `mongodump` (page 1075). You can create backups of your MongoDB cluster, if your backup infrastructure can capture the entire backup in a reasonable amount of time and if you have a storage system that can hold the complete MongoDB data set.

Read *Sharded Cluster Backup Considerations* (page 134) for a high-level overview of important considerations as well as a list of alternate backup tutorials.

Important: By default `mongodump` (page 1075) issue its queries to the non-primary nodes.

Procedure

Capture Data

Note: If you use `mongodump` (page 1075) without specifying a database or collection, `mongodump` (page 1075) will capture collection data *and* the cluster meta-data from the *config servers* (page 498).

You cannot use the `--oplog` option for `mongodump` (page 1075) when capturing data from `mongos` (page 1061). This option is only available when running directly against a *replica set* member.

You can perform a backup of a *sharded cluster* by connecting `mongodump` (page 1075) to a `mongos` (page 1061). Use the following operation at your system's prompt:

```
mongodump --host mongos3.example.net --port 27017
```

`mongodump` (page 1075) will write *BSON* files that hold a copy of data stored in the *sharded cluster* accessible via the `mongos` (page 1061) listening on port `27017` of the `mongos3.example.net` host.

Restore Data Backups created with `mongodump` (page 1075) do not reflect the chunks or the distribution of data in the sharded collection or collections. Like all `mongodump` (page 1075) output, these backups contain separate directories for each database and *BSON* files for each collection in that database.

You can restore `mongodump` (page 1075) output to any MongoDB instance, including a standalone, a *replica set*, or a new *sharded cluster*. When restoring data to sharded cluster, you must deploy and configure sharding before restoring data from the backup. See *Deploy a Sharded Cluster* (page 516) for more information.

Create Backup of a Sharded Cluster with Filesystem Snapshots

Overview

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses file system snapshots to capture a copy of the `mongod` (page 1049) instance. An alternate procedure that uses `mongodump` (page 1075) to create binary database dumps when file-system snapshots are not available. See *Create Backup of a Sharded Cluster with Database Dumps* (page 148) for the alternate procedure.

See *Sharded Cluster Backup Considerations* (page 134) for a full higher level overview backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Procedure

In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using a file-system snapshot tool. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

1. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` (page 1031) method in the `mongo` (page 1066) shell, and see the *Disable the Balancer* (page 543) procedure.

   Warning: It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* may migrate while recording backups.

2. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` (page 1049) instances in as short of an interval as possible.

   To lock or freeze a sharded cluster, you must:

   - use the `db.fsyncLock()` (page 1004) method in the `mongo` (page 1066) shell connected to a single secondary member of the replica set that provides shard `mongod` (page 1049) instance.
   - Shutdown one of the *config servers* (page 498), to prevent all metadata changes during the backup process.

3. Use `mongodump` (page 1075) to backup one of the *config servers* (page 498). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

   Issue this command against one of the config `mongod` (page 1049) instances or via the `mongos` (page 1061):

   ```
   mongodump --db config
   ```

4. Back up the replica set members of the shards that you locked. You may back up the shards in parallel. For each shard, create a snapshot. Use the procedures in *Use Filesystem Snapshots to Backup and Restore MongoDB Databases* (page 138).

5. Unlock all locked replica set members of each shard using the `db.fsyncUnlock()` (page 1004) method in the `mongo` (page 1066) shell.

6. Re-enable the balancer with the `sh.setBalancerState()` (page 1027) method.

   Use the following command sequence when connected to the `mongos` (page 1061) with the `mongo` (page 1066) shell:

```
use config
sh.setBalancerState(true)
```

Create Backup of a Sharded Cluster with Database Dumps

Overview

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses `mongodump` (page 1075) to create dumps of the `mongod` (page 1049) instance. An alternate procedure uses file system snapshots to capture the backup data, and may be more efficient in some situations if your system configuration allows file system backups. See *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 147).

See *Sharded Cluster Backup Considerations* (page 134) for a full higher level overview of backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Procedure

In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using `mongodump` (page 1075) to capture the backup data. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment of time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

1. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` (page 1031) method in the `mongo` (page 1066) shell, and see the *Disable the Balancer* (page 543) procedure.

   Warning: It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* migrate while recording backups.

2. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` (page 1049) instances in as short of an interval as possible.

   To lock or freeze a sharded cluster, you must:

   - Shutdown one member of each replica set.

     Ensure that the *oplog* has sufficient capacity to allow these secondaries to catch up to the state of the primaries after finishing the backup procedure. See *Oplog Size* (page 405) for more information.

   - Shutdown one of the *config servers* (page 498), to prevent all metadata changes during the backup process.

3. Use `mongodump` (page 1075) to backup one of the *config servers* (page 498). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

   Issue this command against one of the config `mongod` (page 1049) instances or via the `mongos` (page 1061):

   ```
   mongodump --journal --db config
   ```

4. Back up the replica set members of the shards that shut down using `mongodump` (page 1075) and specifying the `--dbpath` option. You may back up the shards in parallel. Consider the following invocation:

   ```
   mongodump --journal --dbpath /data/db/ --out /data/backup/
   ```

   You must run this command on the system where the `mongod` (page 1049) ran. This operation will use journaling and create a dump of the entire `mongod` (page 1049) instance with data files stored in `http://docs.mongodb.org/manualdata/db/`. `mongodump` (page 1075) will write the output of this dump to the `http://docs.mongodb.org/manualdata/backup/` directory.

5. Restart all stopped replica set members of each shard as normal and allow them to catch up with the state of the primary.

6. Re-enable the balancer with the `sh.setBalancerState()` (page 1027) method.

   Use the following command sequence when connected to the `mongos` (page 1061) with the `mongo` (page 1066) shell:

   ```
   use config
   sh.setBalancerState(true)
   ```

Restore a Single Shard

Overview

Restoring a single shard from backup with other unaffected shards requires a number of special considerations and practices. This document outlines the additional tasks you must perform when restoring a single shard.

Consider the following resources on backups in general as well as backup and restoration of sharded clusters specifically:

- *Sharded Cluster Backup Considerations* (page 134)
- *Restore Sharded Clusters* (page 150)
- *Backup Strategies for MongoDB Systems* (page 133)

Procedure

Always restore *sharded clusters* as a whole. When you restore a single shard, keep in mind that the *balancer* process might have moved *chunks* to or from this shard since the last backup. If that's the case, you must manually move those chunks, as described in this procedure.

1. Restore the shard as you would any other `mongod` (page 1049) instance. See *Backup Strategies for MongoDB Systems* (page 133) for overviews of these procedures.

2. For all chunks that migrate away from this shard, you do not need to do anything at this time. You do not need to delete these documents from the shard because the chunks are automatically filtered out from queries by `mongos` (page 1061). You can remove these documents from the shard, if you like, at your leisure.

3. For chunks that migrate to this shard after the most recent backup, you must manually recover the chunks using backups of other shards, or some other source. To determine what chunks have moved, view the `changelog` collection in the *Config Database* (page 555).

Restore Sharded Clusters

Overview

The procedure outlined in this document addresses how to restore an entire sharded cluster. For information on related backup procedures consider the following tutorials which describe backup procedures in greater detail:

- *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 147)
- *Create Backup of a Sharded Cluster with Database Dumps* (page 148)

The exact procedure used to restore a database depends on the method used to capture the backup. See the *Backup Strategies for MongoDB Systems* (page 133) document for an overview of backups with MongoDB, as well as *Sharded Cluster Backup Considerations* (page 134) which provides an overview of the high level concepts important for backing up sharded clusters.

Procedure

1. Stop all `mongod` (page 1049) and `mongos` (page 1061) processes.
2. If shard hostnames have changed, you must manually update the `shards` collection in the *Config Database* (page 555) to use the new hostnames. Do the following:
   (a) Start the three *config servers* (page 498) by issuing commands similar to the following, using values appropriate to your configuration:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27019
   ```

   (b) Restore the *Config Database* (page 555) on each config server.
   (c) Start one `mongos` (page 1061) instance.
   (d) Update the *Config Database* (page 555) collection named `shards` to reflect the new hostnames.
3. Restore the following:
   - Data files for each server in each *shard*. Because replica sets provide each production shard, restore all the members of the replica set or use the other standard approaches for restoring a replica set from backup. See the *Restore a Snapshot* (page 140) and *Restore a Database with mongorestore* (page 137) sections for details on these procedures.
   - Data files for each *config server* (page 498), if you have not already done so in the previous step.
4. Restart all the `mongos` (page 1061) instances.
5. Restart all the `mongod` (page 1049) instances.
6. Connect to a `mongos` (page 1061) instance from a `mongo` (page 1066) shell and use the `db.printShardingStatus()` (page 1010) method to ensure that the cluster is operational, as follows:

   ```
   db.printShardingStatus()
   show collections
   ```

Schedule Backup Window for Sharded Clusters

Overview

In a *sharded cluster*, the balancer process is responsible for distributing sharded data around the cluster, so that each *shard* has roughly the same amount of data.

However, when creating backups from a sharded cluster it is important that you disable the balancer while taking backups to ensure that no chunk migrations affect the content of the backup captured by the backup procedure. Using the procedure outlined in the section *Disable the Balancer* (page 543) you can manually stop the balancer process temporarily. As an alternative you can use this procedure to define a balancing window so that the balancer is always disabled during your automated backup operation.

Procedure

If you have an automated backup schedule, you can disable all balancing operations for a period of time. For instance, consider the following command:

```
use config
db.settings.update( { _id : "balancer" }, { $set : { activeWindow : { start : "6:00", stop : "23:00"
```

This operation configures the balancer to run between 6:00am and 11:00pm, server time. Schedule your backup operation to run *and complete* outside of this time. Ensure that the backup can complete outside the window when the balancer is running *and* that the balancer can effectively balance the collection among the shards in the window allotted to each.

10.3 Data Center Awareness

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

MongoDB also supports segregation based on functional parameters, to ensure that certain `mongod` (page 1049) instances are only used for reporting workloads or that certain high-frequency portions of a sharded collection only exist on specific shards.

Consider the following documents:

10.3.1 Operational Segregation in MongoDB Operations and Deployments

Operational Overview

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

This capability provides "data center awareness," which allows applications to target MongoDB deployments with consideration of the physical location of the `mongod` (page 1049) instances. MongoDB supports segmentation of operations across different dimensions, which may include multiple data centers and geographical regions in multi-data center deployments, racks, networks, or power circuits in single data center deployments.

MongoDB also supports segregation of database operations based on functional or operational parameters, to ensure that certain `mongod` (page 1049) instances are only used for reporting workloads or that certain high-frequency portions of a sharded collection only exist on specific shards.

Specifically, with MongoDB, you can:

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

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

- *Read Preferences* (page 398), which controls how drivers help applications target read operations to members of a replica set.
- *Write Concerns* (page 395), which controls how MongoDB ensures that write operations propagate to members of a replica set.
- *Replica Set Tags* (page 444), which control how applications create and interact with custom groupings of replica set members to create custom application-specific read preferences and write concerns.
- *Tag Aware Sharding* (page 547), which allows MongoDB administrators to define an application-specific balancing policy, to control how documents belonging to specific ranges of a shard key distribute to shards in the *sharded cluster*.

See also:

Before adding operational segregation features to your application and MongoDB deployment, become familiar with all documentation of *replication* (page 367), :and doc:*sharding </sharding>*.

10.3.2 Tag Aware Sharding

For sharded clusters, MongoDB makes it possible to associate specific ranges of a *shard key* with a specific *shard* or subset of shards. This association dictates the policy of the cluster balancer process as it balances the *chunks* around the cluster. This capability enables the following deployment patterns:

- isolating a specific subset of data on specific set of shards.
- controlling the balancing policy so that, in a geographically distributed cluster, the most relevant portions of the data set reside on the shards with the greatest proximity to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

Note: Shard key range tags are entirely distinct from *replica set member tags* (page 401).

Hash-based sharding does not support tag-aware sharding.

Behavior and Operations

Tags in a sharded cluster are pieces of metadata that dictate the policy and behavior of the cluster *balancer*. Using tags, you may associate individual shards in a cluster with one or more tags. Then, you can assign this tag string to a range of *shard key* values for a sharded collection. When migrating a chunk, the balancer will select a destination shard based on the configured tag ranges.

The balancer migrates chunks in tagged ranges to shards with those tags, if tagged shards are not balanced. [5]

[5] To migrate chunks in a tagged environment, the balancer selects a target shard with a tag range that has an *upper* bound that is *greater than* the migrating chunk's *lower* bound. If a shard with a matching tagged range exists, the balancer will migrate the chunk to that shard.

Note: Because a single chunk may span different tagged shard key ranges, the balancer may migrate chunks to tagged shards that contain values that exceed the upper bound of the selected tag range.

Example

Given a sharded collection with two configured tag ranges, such that:

- *Shard key* values between `100` and `200` have tags to direct corresponding chunks to shards tagged `NYC`.
- Shard Key values between `200` and `300` have tags to direct corresponding chunks to shards tagged `SFO`.

In this cluster, the balancer will migrate a chunk with shard key values ranging between `150` and `220` to a shard tagged `NYC`, since `150` is closer to `200` than `300`.

After configuring tags on the shards and ranges of the shard key, the cluster may take some time to reach the proper distribution of data, depending on the division of chunks (i.e. splits) and the current distribution of data in the cluster. Once configured, the balancer will respect tag ranges during future *balancing rounds* (page 510).

See also:

Administer and Manage Shard Tags (page 530)

10.3.3 Administer and Manage Shard Tags

In a sharded cluster, you can use tags to associate specific ranges of a *shard key* with a specific *shard* or subset of shards.

Tag a Shard

Associate tags with a particular shard using the `sh.addShardTag()` (page 1023) method when connected to a `mongos` (page 1061) instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag `NYC` to two shards, and the tags `SFO` and `NRT` to a third shard:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

You may remove tags from a particular shard using the `sh.removeShardTag()` (page 1027) method when connected to a `mongos` (page 1061) instance, as in the following example, which removes the `NRT` tag from a shard:

```
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range

To assign a tag to a range of shard keys use the `sh.addTagRange()` (page 1023) method when connected to a `mongos` (page 1061) instance. Any given shard key range may only have *one* assigned tag. You cannot overlap defined ranges, or tag the same range more than once.

Example

Given a collection named `users` in the `records` database, sharded by the `zipcode` field. The following operations assign:

- two ranges of zip codes in Manhattan and Brooklyn the `NYC` tag
- one range of zip codes in San Francisco the `SFO` tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Note: Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Remove a Tag From a Shard Key Range

The `mongod` (page 1049) does not provide a helper for removing a tag range. You may delete tag assignment from a shard key range by removing the corresponding document from the `tags` (page 560) collection of the `config` database.

Each document in the `tags` (page 560) holds the *namespace* of the sharded collection and a minimum shard key value.

Example

The following example removes the `NYC` tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

View Existing Shard Tags

The output from `sh.status()` (page 1029) lists tags associated with a shard, if any, for each shard. A shard's tags exist in the shard's document in the `shards` (page 560) collection of the `config` database. To return all shards with a specific tag, use a sequence of operations that resemble the following, which will return only those shards tagged with `NYC`:

```
use config
db.shards.find({ tags: "NYC" })
```

You can find tag ranges for all *namespaces* in the `tags` (page 560) collection of the `config` database. The output of `sh.status()` (page 1029) displays all tag ranges. To return all shard key ranges tagged with `NYC`, use the following sequence of operations:

```
use config
db.tags.find({ tags: "NYC" })
```

- *Deploy a Geographically Distributed Replica Set* (page 420)

Additionally, consider the *Write Concern* (page 395) and *Read Preference* (page 398) documents, which addresses capabilities related to data center awareness.

10.4 Journaling

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 53) durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal. If MongoDB should terminate or encounter an error before it can write the changes from the journal to the data files, MongoDB can re-apply the write operation and maintain a consistent state.

Without a journal, if `mongod` (page 1049) exits unexpectedly, you must assume your data is in an inconsistent state, and you must run either *repair* (page 144) or, preferably, *resync* (page 443) from a clean member of the replica set.

With journaling enabled, if `mongod` (page 1049) stops unexpectedly, the program can recover everything written to the journal, and the data remains in a consistent state. By default, the greatest extent of lost writes, i.e., those not made to the journal, are those made in the last 100 milliseconds. See `journalCommitInterval` (page 1119) for more information on the default.

With journaling, if you want a data set to reside entirely in RAM, you need enough RAM to hold the data set plus the "write working set." The "write working set" is the amount of unique data you expect to see written between re-mappings of the private view. For information on views, see *Storage Views used in Journaling* (page 157).

Important: Changed in version 2.0: For 64-bit builds of `mongod` (page 1049), journaling is enabled by default. For other platforms, see `journal` (page 1119).

10.4.1 Procedures

Enable Journaling

Changed in version 2.0: For 64-bit builds of `mongod` (page 1049), journaling is enabled by default.

To enable journaling, start `mongod` (page 1049) with the `--journal` command line option.

If no journal files exist, when `mongod` (page 1049) starts, it must preallocate new journal files. During this operation, the `mongod` (page 1049) is not listening for connections until preallocation completes: for some systems this may take a several minutes. During this period your applications and the `mongo` (page 1066) shell are not available.

Disable Journaling

Warning: Do not disable journaling on production systems. If your `mongod` (page 1049) instance stops without shutting down cleanly unexpectedly for any reason, (e.g. power failure) and you are not running with journaling, then you must recover from an unaffected *replica set* member or backup, as described in *repair* (page 144).

To disable journaling, start `mongod` (page 1049) with the `--nojournal` command line option.

Get Commit Acknowledgment

You can get commit acknowledgment with the `getLastError` (page 861) command and the `j` option. For details, see *Write Concern Reference* (page 57).

Avoid Preallocation Lag

To avoid *preallocation lag* (page 157), you can preallocate files in the journal directory by copying them from another instance of `mongod` (page 1049).

Preallocated files do not contain data. It is safe to later remove them. But if you restart mongod (page 1049) with journaling, mongod (page 1049) will create them again.

Example

The following sequence preallocates journal files for an instance of mongod (page 1049) running on port 27017 with a database path of http://docs.mongodb.org/manualdata/db.

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

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

```
mkdir ~/tmpDbpath
```

2. Create a set of journal files by staring a mongod (page 1049) instance that uses the temporary directory:

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

3. When you see the following log output, indicating mongod (page 1049) has the files, press CONTROL+C to stop the mongod (page 1049) instance:

```
web admin interface listening on port 11000
```

4. Preallocate journal files for the new instance of mongod (page 1049) by moving the journal files from the data directory of the existing instance to the data directory of the new instance:

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

5. Start the new mongod (page 1049) instance:

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

Monitor Journal Status

Use the following commands and methods to monitor journal status:

- serverStatus (page 919)

  The serverStatus (page 919) command returns database status information that is useful for assessing performance.

- journalLatencyTest (page 942)

  Use journalLatencyTest (page 942) to measure how long it takes on your volume to write to the disk in an append-only fashion. You can run this command on an idle system to get a baseline sync time for journaling. You can also run this command on a busy system to see the sync time on a busy system, which may be higher if the journal directory is on the same volume as the data files.

  The journalLatencyTest (page 942) command also provides a way to check if your disk drive is buffering writes in its local cache. If the number is very low (i.e., less than 2 milliseconds) and the drive is non-SSD, the drive is probably buffering writes. In that case, enable cache write-through for the device in your operating system, unless you have a disk controller card with battery backed RAM.

Change the Group Commit Interval

Changed in version 2.0.

You can set the group commit interval using the `--journalCommitInterval` command line option. The allowed range is `2` to `300` milliseconds.

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

Recover Data After Unexpected Shutdown

On a restart after a crash, MongoDB replays all journal files in the journal directory before the server becomes available. If MongoDB must replay journal files, `mongod` (page 1049) notes these events in the log output.

There is no reason to run `repairDatabase` (page 895) in these situations.

10.4.2 Journaling Internals

When running with journaling, MongoDB stores and applies *write operations* (page 53) in memory and in the journal before the changes are in the data files.

Journal Files

With journaling enabled, MongoDB creates a journal directory within the directory defined by `dbpath` (page 1118), which is `http://docs.mongodb.org/manualdata/db` by default. The journal directory holds journal files, which contain write-ahead redo logs. The directory also holds a last-sequence-number file. A clean shutdown removes all the files in the journal directory.

Journal files are append-only files and have file names prefixed with `j._`. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. Once MongoDB applies all the write operations in the journal files, it deletes these files. Unless you write *many* bytes of data per-second, the journal directory should contain only two or three journal files.

To limit the size of each journal file to 128 megabytes, use the `smallfiles` (page 1122) run time option when starting `mongod` (page 1049).

To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different system.

Important: If you place the journal on a different filesystem from your data files you *cannot* use a filesystem snapshot to capture consistent backups of a `dbpath` (page 1118) directory.

Note: Depending on your file system, you might experience a preallocation lag the first time you start a `mongod` (page 1049) instance with journaling enabled.

MongoDB may preallocate journal files if the `mongod` (page 1049) process determines that it is more efficient to preallocate journal files than create new journal files as needed. The amount of time required to pre-allocate lag might last several minutes, during which you will not be able to connect to the database. This is a one-time preallocation and does not occur with future invocations.

To avoid preallocation lag, see *Avoid Preallocation Lag* (page 155).

Storage Views used in Journaling

Journaling adds three storage views to MongoDB.

The `shared view` stores modified data for upload to the MongoDB data files. The `shared view` is the only view with direct access to the MongoDB data files. When running with journaling, `mongod` (page 1049) asks the operating system to map your existing on-disk data files to the `shared view` memory view. The operating system maps the files but does not load them. MongoDB later loads data files to `shared view` as needed.

The `private view` stores data for use in *read operations* (page 41). MongoDB maps `private view` to the `shared view` and is the first place MongoDB applies new *write operations* (page 53).

The journal is an on-disk view that stores new write operations after MongoDB applies the operation to the `private cache` but before applying them to the data files. The journal provides durability. If the `mongod` (page 1049) instance were to crash without having applied the writes to the data files, the journal could replay the writes to the `shared view` for eventual upload to the data files.

How Journaling Records Write Operations

MongoDB copies the write operations to the journal in batches called group commits. See `journalCommitInterval` (page 1119) for more information on the default commit interval. These "group commits" help minimize the performance impact of journaling.

Journaling stores raw operations that allow MongoDB to reconstruct the following:

- document insertion/updates
- index modifications
- changes to the namespace files

As *write operations* (page 53) occur, MongoDB writes the data to the `private view` in RAM and then copies the write operations in batches to the journal. The journal stores the operations on disk to ensure durability. MongoDB adds the operations as entries on the journal's forward pointer. Each entry describes which bytes the write operation changed in the data files.

MongoDB next applies the journal's write operations to the `shared view`. At this point, the `shared view` becomes inconsistent with the data files.

At default intervals of 60 seconds, MongoDB asks the operating system to flush the `shared view` to disk. This brings the data files up-to-date with the latest write operations.

When MongoDB flushes write operations to the data files, MongoDB removes the write operations from the journal's behind pointer. The behind pointer is always far back from advanced pointer.

As part of journaling, MongoDB routinely asks the operating system to remap the `shared view` to the `private view`, for consistency.

Note: The interaction between the `shared view` and the on-disk data files is similar to how MongoDB works *without* journaling, which is that MongoDB asks the operating system to flush in-memory changes back to the data files every 60 seconds.

10.5 Monitoring for MongoDB

Monitoring is a critical component of all database administration. A firm grasp of MongoDB's reporting will allow you to assess the state of your database and maintain your deployment without crisis. Additionally, a sense of MongoDB's normal operational parameters will allow you to diagnose 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 an 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 Management Service (MMS) and the MMS documentation for more information.

10.5.1 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* (page 833) 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` (page 1103) tracks and reports the current read and write activity of a MongoDB instance. `mongotop` (page 1103) provides per-collection visibility into use. Use `mongotop` (page 1103) to verify that activity and use match expectations. See the *mongotop manual* (page 1103) for details.

`mongostat`

`mongostat` (page 1098) captures and returns counters of database operations. `mongostat` (page 1098) 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` (page 1098) to understand the distribution of operation types and to inform capacity planning. See the *mongostat manual* (page 1097) 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 1121) 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

MongoDB 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 to modify the behavior of your application in response to the activity of your instance.

`serverStatus`

Access *serverStatus data* (page 919) by way of the `serverStatus` (page 919) 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` (page 919).

See also:

`db.serverStatus()` (page 1012) and *serverStatus data* (page 919).

replSetGetStatus

View the *replSetGetStatus data* (page 865) with the `replSetGetStatus` (page 865) command (`rs.status()` (page 1017) 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 904) is accessible by way of the `dbStats` (page 904) command (`db.stats()` (page 1013) 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 900) is accessible using the `collStats` (page 900) command (`db.printCollectionStats()` (page 1010) from the shell). It provides statistics that resemble `dbStats` (page 904) 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.

Introspection Tools

In addition to status reporting, MongoDB provides a number of introspection tools that you can use to diagnose and analyze performance and operational conditions. Consider the following documentation:

- `diagLogging` (page 906)
- *Analyze Performance of Database Operations* (page 175)
- *Database Profiler Output* (page 1134)
- *db.currentOp()* (page 998)

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	Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	gmond_python_modules	Parses output from the serverStatus (page 919) and replSetGetStatus (page 865) 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, written in Python.
Zabbix	mikoomi-mongodb	Monitors availability, resource utilization, health, performance and other important metrics.

Also consider dex, an 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.

10.5.2 Process Logging

During normal operation, `mongod` (page 1049) and `mongos` (page 1061) instances report information that reflect current operation to standard output, or a log file. The following runtime settings control these options.

- `quiet` (page 1123). Limits the amount of information written to the log or output.
- `verbose` (page 1115). Increases the amount of information written to the log or output.

  You can also specify this as `v` (as in `-v`.) Set multiple `v`, as in `vvvv = True` for higher levels of verbosity. You can also change the verbosity of a running `mongod` (page 1049) or `mongos` (page 1061) instance with the `setParameter` (page 894) command.
- `logpath` (page 1116). Enables logging to a file, rather than standard output. Specify the full path to the log file to this setting.
- `logappend` (page 1117). Adds information to a log file instead of overwriting the file.

Note: You can specify these configuration operations as the command line arguments to *mongod* (page 1049) or *mongos* (page 1061)

Additionally, the following *database commands* affect logging:

- `getLog` (page 916). Displays recent messages from the `mongod` (page 1049) process log.

- `logRotate` (page 897). Rotates the log files for `mongod` (page 1049) processes only. See *Rotate Log Files* (page 190).

10.5.3 Diagnosing Performance Issues

Degraded performance in MongoDB can be the result of an array of causes, and is typically a function of the relationship among 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 is 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 922) section of the `serverStatus` (page 919) response to assess if the lock has been a challenge to your performance. If `globalLock.currentQueue.total` (page 923) 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 affect performance.

If `globalLock.totalTime` (page 922) is high in context of `uptime` (page 920) then the database has existed in a lock state for a significant amount of time. If `globalLock.ratio` (page 923) 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 162) 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 923) to better understand MongoDB's memory utilization. Check the resident memory use (i.e. `mem.resident` (page 924):) 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 924).) 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 925) value in the `serverStatus` (page 919) 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` (page 1049) 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 919) document:

- `globalLock.activeClients` (page 923) contains a counter of the total number of clients with active operations in progress or queued.
- `connections` (page 924) is a container for the following two fields:
  - `current` (page 924) the total number of current clients that connect to the database instance.
  - `available` (page 924) 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 `/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` (page 1049) 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` (page 907) value using the following command in the `mongo` (page 1066) shell:

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

See

The documentation of `db.setProfilingLevel()` (page 1012) 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` (page 1049) 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 1121) setting 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 1121) at runtime, as an argument to the `db.setProfilingLevel()` (page 1012) operation.

Additionally, `mongod` (page 1049) records all "slow" queries to its `log` (page 1116), as defined by `slowms` (page 1121). The data in `system.profile` does not persist between `mongod` (page 1049) restarts.

You can view the profiler's output by issuing the `show profile` command in the `mongo` (page 1066) 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 1121) threshold.

See also:

Optimization Strategies for MongoDB (page 576) addresses strategies that may improve the performance of your database queries and operations.

10.5.4 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 MongoDB will have to perform an initial sync on the secondary, copying all data from the *primary* and rebuilding 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 455).

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` (page 865) or the following helper in the shell:

```
rs.status()
```

See the *replSetGetStatus* (page 865) document for a more in depth overview view of this output. In general watch the value of `optimeDate` (page 866). 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* argument to the `mongod` (page 1049) command, or preferably the `oplogSize` (page 1124) in the MongoDB configuration file. If you do not specify this on the command line before running with the *--replSet* option, `mongod` (page 1049) will create a 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 439)

10.5.5 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* (page 485) 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 1061) instances become unavailable. However, clusters remain accessible from already-running `mongos` (page 1061) 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 1061) instances can restart.

Balancing and Chunk Distribution

The most effective *sharded cluster* deployments require that *chunks* are evenly balanced among 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()` (page 1010) or `sh.status()` (page 1029) command to the `mongos` (page 1061) by way of the `mongo` (page 1066) 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 1061) instance using the `mongo` (page 1066) 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 1061), 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.

10.6 Import and Export MongoDB Data

This document provides an overview of the import and export programs included in the MongoDB distribution. These tools are useful when you want to backup or export a portion of your data without capturing the state of the entire database, or for simple data ingestion cases. For more complex data migration tasks, you may want to write your own import and export scripts using a client *driver* to interact with the database itself. For disaster recovery protection and routine database backup operation, use full *database instance backups* (page 133).

> **Warning:** Because these tools primarily operate by interacting with a running `mongod` (page 1049) instance, they can impact the performance of your running database.
>
> Not only do these processes create traffic for a running database instance, they also force the database to read all data through memory. When MongoDB reads infrequently used data, it can supplant more frequently accessed data, causing a deterioration in performance for the database's regular workload.
>
> `mongoimport` (page 1089) and `mongoexport` (page 1093) do not reliably preserve all rich *BSON* data types, because *BSON* is a superset of *JSON*. Thus, `mongoimport` (page 1089) and `mongoexport` (page 1093) cannot represent *BSON* data accurately in *JSON*. As a result data exported or imported with these tools may lose some measure of fidelity. See *MongoDB Extended JSON* (page 1147) for more information about MongoDB Extended JSON.

See also:

See the "*Backup Strategies for MongoDB Systems* (page 133)" document for more information on backing up MongoDB instances. Additionally, consider the following references for commands addressed in this document:

- *mongoexport* (page 1093)
- *mongorestore* (page 1079)
- *mongodump* (page 1075)

If you want to transform and process data once you've imported it in MongoDB consider the documents in the *Aggregation* (page 245) section, including:

- *Map-Reduce* (page 291) and
- *Aggregation Framework* (page 247).

10.6.1 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 (page 1147)

10.6.2 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 Strategies for MongoDB Systems* (page 133)" 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.

10.6.3 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 1089) and *mongoexport* (page 1093) 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` (page 883), `clone` (page 885), or `cloneCollection` (page 886) commands, which may be more suited to this task. The `mongo` (page 1066) shell provides the `db.copyDatabase()` (page 996) 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` (page 1093) utility you can create a backup file. In the most simple invocation, the command takes the following form:

```
mongoexport --collection collection --out collection.json
```

This will export all documents in the collection named `collection` into the file `collection.json`. Without the output specification (i.e. "*--out collection.json*",) `mongoexport` (page 1093) writes output to standard output (i.e. "stdout.") You can further narrow the results by supplying a query filter using the "*--query*" and limit results to a single database using the "*--db*" option. For instance:

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

This command returns all documents in the `sales` database's `contacts` collection, with a field named `field` with a value of `1`. Enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment. The resulting documents will return on standard output.

By default, `mongoexport` (page 1093) returns one *JSON document* per MongoDB document. Specify the "*--jsonArray*" argument to return the export as a single *JSON* array. Use the "*--csv*" file to return the result in CSV (comma separated values) format.

If your `mongod` (page 1049) instance is not running, you can use the "*--dbpath*" option to specify the location to your MongoDB instance's database files. See the following example:

```
mongoexport --db sales --collection contacts --dbpath /srv/MongoDB/
```

This reads the data files directly. This locks the data directory to prevent conflicting writes. The `mongod` (page 1049) process must *not* be running or attached to these data files when you run `mongoexport` (page 1093) in this configuration.

The "*--host*" and "*--port*" options allow you to specify a non-local host to connect to capture the export. Consider the following example:

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

On any `mongoexport` (page 1093) command you may, as above specify username and password credentials as above.

Collection Import with `mongoimport`

To restore a backup taken with `mongoexport` (page 1093). Most of the arguments to `mongoexport` (page 1093) also exist for `mongoimport` (page 1089). Consider the following command:

```
mongoimport --collection collection --file collection.json
```

This imports the contents of the file `collection.json` into the collection named `collection`. If you do not specify a file with the "*--file*" option, `mongoimport` (page 1089) accepts input over standard input (e.g. "stdin.")

If you specify the "*--upsert*" option, all of mongoimport (page 1089) operations will attempt to update existing documents in the database and insert other documents. This option will cause some performance impact depending on your configuration.

You can specify the database option *--db* to import these documents to a particular database. If your MongoDB instance is not running, use the "*--dbpath*" option to specify the location of your MongoDB instance's database files. Consider using the "*--journal*" option to ensure that mongoimport (page 1089) records its operations in the journal. The mongod process must *not* be running or attached to these data files when you run mongoimport (page 1089) in this configuration.

Use the "*--ignoreBlanks*" option to ignore blank fields. For *CSV* and *TSV* imports, this option provides the desired functionality in most cases: it avoids inserting blank fields in MongoDB documents.

CHAPTER 11

Processes

11.1 MongoDB Tutorials

This page lists the tutorials available as part of the *MongoDB Manual* (page 1). In addition to these documents, you can refer to the introductory *MongoDB Tutorial* (page 29). If there is a process or pattern that you would like to see included here, please open a Jira Case.

11.1.1 Getting Started

- *Install MongoDB on Linux* (page 11)
- *Install MongoDB on Red Hat Enterprise, CentOS, or Fedora Linux* (page 3)
- *Install MongoDB on Debian* (page 9)
- *Install MongoDB on Ubuntu* (page 6)
- *Install MongoDB on OS X* (page 13)
- *Install MongoDB on Windows* (page 16)
- *Getting Started with MongoDB* (page 29)
- *Generate Test Data* (page 34)

11.1.2 Administration

Replica Sets

- *Deploy a Replica Set* (page 416)
- *Convert a Standalone to a Replica Set* (page 426)
- *Add Members to a Replica Set* (page 427)
- *Remove Members from Replica Set* (page 430)
- *Replace a Replica Set Member* (page 430)
- *Adjust Priority for Replica Set Member* (page 431)
- *Resync a Member of a Replica Set* (page 443)
- *Deploy a Geographically Distributed Replica Set* (page 420)

Sharding

Basic Operations

Security

11.1.3 Development Patterns

11.1.4 Application Development

11.1.5 Text Search Patterns

11.1.6 Data Modeling Patterns

11.1.7 MongoDB Use Case Studies

11.2 Analyze Performance of Database Operations

The database profiler collects fine grained data about MongoDB write operations, cursors, database commands on a running `mongod` (page 1049) instance. You can enable profiling on a per-database or per-instance basis. The *profiling level* (page 175) is also configurable when enabling profiling.

The database profiler writes all the data it collects to the `system.profile` (page 1133) collection, which is a *capped collection* (page 578). See *Database Profiler Output* (page 1134) for overview of the data in the `system.profile` (page 1133) documents created by the profiler.

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

- *Database Profiler Output* (page 1134)
- *Profile Command* (page 907)
- *db.currentOp()* (page 998)

11.2.1 Profiling Levels

The following profiling levels are available:

- `0` - the profiler is off, does not collect any data.
- `1` - collects profiling data for slow operations only. By default slow operations are those slower than 100 milliseconds.

  You can modify the threshold for "slow" operations with the `slowms` (page 1121) runtime option or the `setParameter` (page 894) command. See the *Specify the Threshold for Slow Operations* (page 176) section for more information.
- `2` - collects profiling data for all database operations.

11.2.2 Enable Database Profiling and Set the Profiling Level

You can enable database profiling from the `mongo` (page 1066) shell or through a driver using the `profile` (page 907) command. This section will describe how to do so from the `mongo` (page 1066) shell. See your *driver documentation* (page 575) if you want to control the profiler from within your application.

When you enable profiling, you also set the *profiling level* (page 175). The profiler records data in the `system.profile` (page 1133) collection. MongoDB creates the `system.profile` (page 1133) collection in a database after you enable profiling for that database.

To enable profiling and set the profiling level, issue use the `db.setProfilingLevel()` (page 1012) helper in the `mongo` (page 1066) shell, passing the profiling level as a parameter. For example, to enable profiling for all database operations, consider the following operation in the `mongo` (page 1066) shell:

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

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

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

To verify the new setting, see the *Check Profiling Level* (page 176) section.

Specify the Threshold for Slow Operations

The threshold for slow operations applies to the entire `mongod` (page 1049) instance. When you change the threshold, you change it for all databases on the instance.

Important: Changing the slow operation threshold for the database profiler also affects the profiling subsystem's slow operation threshold for the entire `mongod` (page 1049) instance. Always set the threshold to the highest useful value.

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

To change the threshold, pass two parameters to the `db.setProfilingLevel()` (page 1012) helper in the `mongo` (page 1066) shell. The first parameter sets the profiling level for the current database, and the second sets the default slow operation threshold *for the entire* `mongod` (page 1049) *instance*.

For example, the following command sets the profiling level for the current database to `0`, which disables profiling, and sets the slow-operation threshold for the `mongod` (page 1049) instance to 20 milliseconds. Any database on the instance with a profiling level of `1` will use this threshold:

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

Check Profiling Level

To view the *profiling level* (page 175), issue the following from the `mongo` (page 1066) shell:

```
db.getProfilingStatus()
```

The shell returns a document similar to the following:

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

The `was` field indicates the current level of profiling.

The `slowms` field indicates how long an operation must exist in milliseconds for an operation to pass the "slow" threshold. MongoDB will log operations that take longer than the threshold if the profiling level is `1`. This document returns the profiling level in the `was` field. For an explanation of profiling levels, see *Profiling Levels* (page 175).

To return only the profiling level, use the `db.getProfilingLevel()` (page 1006) helper in the `mongo` (page 1066) as in the following:

```
db.getProfilingLevel()
```

Disable Profiling

To disable profiling, use the following helper in the `mongo` (page 1066) shell:

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

Enable Profiling for an Entire `mongod` Instance

For development purposes in testing environments, you can enable database profiling for an entire `mongod` (page 1049) instance. The profiling level applies to all databases provided by the `mongod` (page 1049) instance.

To enable profiling for a `mongod` (page 1049) instance, pass the following parameters to `mongod` (page 1049) at startup or within the *configuration file* (page 1115):

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

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

See also:

`profile` (page 1121) and `slowms` (page 1121).

Database Profiling and Sharding

You *cannot* enable profiling on a `mongos` (page 1061) instance. To enable profiling in a shard cluster, you must enable profiling for each `mongod` (page 1049) instance in the cluster.

11.2.3 View Profiler Data

The database profiler logs information about database operations in the `system.profile` (page 1133) collection.

To view profiling information, query the `system.profile` (page 1133) collection. To view example queries, see *Profiler Overhead* (page 178)

For an explanation of the output data, see *Database Profiler Output* (page 1134).

Example Profiler Data Queries

This section displays example queries to the `system.profile` (page 1133) collection. For an explanation of the query output, see *Database Profiler Output* (page 1134).

To return the most recent 10 log entries in the `system.profile` (page 1133) collection, run a query similar to the following:

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

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

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

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

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

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

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

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

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

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

```
db.system.profile.find(
                       {
                         ts : {
                               $gt : new ISODate("2011-07-12T03:00:00Z") ,
                               $lt : new ISODate("2011-07-12T03:40:00Z")
                              }
                       },
                       { user : 0 }
                      ).sort( { millis : -1 } )
```

Show the Five Most Recent Events

On a database that has profiling enabled, the `show profile` helper in the `mongo` (page 1066) shell displays the 5 most recent operations that took at least 1 millisecond to execute. Issue `show profile` from the `mongo` (page 1066) shell, as follows:

```
show profile
```

11.2.4 Profiler Overhead

When enabled, profiling has a minor effect on performance. The `system.profile` (page 1133) collection is a *capped collection* with a default size of 1 megabyte. A collection of this size can typically store several thousand profile documents, but some application may use more or less profiling data per operation.

To change the size of the `system.profile` (page 1133) collection, you must:

1. Disable profiling.
2. Drop the `system.profile` (page 1133) collection.
3. Create a new `system.profile` (page 1133) collection.
4. Re-enable profiling.

For example, to create a new `system.profile` (page 1133) collections that's `4000000` bytes, use the following sequence of operations in the `mongo` (page 1066) shell:

```
db.setProfilingLevel(0)

db.system.profile.drop()

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

db.setProfilingLevel(1)
```

11.3 Use Database Commands

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

See *Database Commands* (page 833) for list of all commands sorted by function, and *Database Commands* (page 833) for a list of all commands sorted alphabetically.

11.3.1 Database Command Form

You specify a command first by constructing a standard *BSON* document whose first key is the name of the command. For example, specify the `isMaster` (page 871) command using the following *BSON* document:

```
{ isMaster: 1 }
```

11.3.2 Issue Commands

The `mongo` (page 1066) shell provides a helper method for running commands called `db.runCommand()` (page 1012). The following operation in `mongo` (page 1066) runs the above command:

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

Many *drivers* (page 575) provide an equivalent for the `db.runCommand()` (page 1012) method. Internally, running commands with `db.runCommand()` (page 1012) is equivalent to a special query against the *$cmd* collection.

Many common commands have their own shell helpers or wrappers in the `mongo` (page 1066) shell and drivers, such as the `db.isMaster()` (page 1008) method in the `mongo` (page 1066) JavaScript shell.

11.3.3 `admin` Database Commands

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

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

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

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

11.3.4 Command Responses

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

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

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

11.4 Connect to MongoDB with SSL

This document outlines the use and operation of MongoDB's SSL support. SSL allows MongoDB clients to support encrypted connections to `mongod` (page 1049) instances.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL, you must either build MongoDB locally passing the "`--ssl`" option to `scons` or use MongoDB Enterprise.

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.

11.4.1 Configure mongod and mongos for SSL

Combine SSL Certificate and Key File

Before you can use SSL, you must have a .pem file that contains the public key certificate and private key. MongoDB can use any valid SSL certificate. To generate a self-signed certificate and private key, use a command that resembles the following:

```
cd /etc/ssl/
openssl req -new -x509 -days 365 -nodes -out mongodb-cert.crt -keyout mongodb-cert.key
```

This operation generates a new, self-signed certificate with no passphrase that is valid for 365 days. Once you have the certificate, concatenate the certificate and private key to a .pem file, as in the following example:

```
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

Set Up mongod and mongos with SSL Certificate and Key

To use SSL in your MongoDB deployment, include the following run-time options with mongod (page 1049) and mongos (page 1061):

- sslOnNormalPorts (page 1127)
- sslPEMKeyFile (page 1127) with the .pem file that contains the SSL certificate and key.

Consider the following syntax for mongod (page 1049):

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem>
```

For example, given an SSL certificate located at /etc/ssl/mongodb.pem, configure mongod (page 1049) to use SSL encryption for all connections with the following command:

```
mongod --sslOnNormalPorts --sslPEMKeyFile /etc/ssl/mongodb.pem
```

Note:

- Specify <pem> with the full path name to the certificate.
- If the private key portion of the <pem> is encrypted, specify the encryption password with the sslPEMKeyPassword (page 1127) option.
- You may also specify these options in the *configuration file* (page 1115), as in the following example:

```
sslOnNormalPorts = true
sslPEMKeyFile = /etc/ssl/mongodb.pem
```

To connect, to mongod (page 1049) and mongos (page 1061) instances using SSL, the mongo (page 1066) shell and MongoDB tools must include the --ssl option. See *SSL Configuration for Clients* (page 182) for more information on connecting to mongod (page 1049) and mongos (page 1061) running with SSL.

Set Up mongod and mongos with Certificate Validation

To set up mongod (page 1049) or mongos (page 1061) for SSL encryption using an SSL certificate signed by a certificate authority, include the following run-time options during startup:

- sslOnNormalPorts (page 1127)
- sslPEMKeyFile (page 1127) with the name of the .pem file that contains the signed SSL certificate and key.

- sslCAFile (page 1128) with the name of the .pem file that contains the root certificate chain from the Certificate Authority.

Consider the following syntax for mongod (page 1049):

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem> --sslCAFile <ca>
```

For example, given a signed SSL certificate located at /etc/ssl/mongodb.pem and the certificate authority file at /etc/ssl/ca.pem, you can configure mongod (page 1049) for SSL encryption as follows:

```
mongod --sslOnNormalPorts --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCAFile /etc/ssl/ca.pem
```

Note:

- Specify the <pem> file and the <ca> file with either the full path name or the relative path name.
- If the <pem> is encrypted, specify the encryption password with the sslPEMKeyPassword (page 1127) option.
- You may also specify these options in the *configuration file* (page 1115), as in the following example:

  ```
  sslOnNormalPorts = true
  sslPEMKeyFile = /etc/ssl/mongodb.pem
  sslCAFile = /etc/ssl/ca.pem
  ```

To connect, to mongod (page 1049) and mongos (page 1061) instances using SSL, the mongo (page 1066) tools must include the both the *--ssl* and *--sslPEMKeyFile* option. See *SSL Configuration for Clients* (page 182) for more information on connecting to mongod (page 1049) and mongos (page 1061) running with SSL.

Block Revoked Certificates for Clients

To prevent clients with revoked certificates from connecting, include the sslCRLFile (page 1128) to specify a .pem file that contains revoked certificates.

For example, the following mongod (page 1049) with SSL configuration includes the sslCRLFile (page 1128) setting:

```
mongod --sslOnNormalPorts --sslCRLFile /etc/ssl/ca-crl.pem --sslPEMKeyFile /etc/ssl/mongodb.pem --ssl
```

Clients with revoked certificates in the /etc/ssl/ca-crl.pem will not be able to connect to this mongod (page 1049) instance.

Validate Only if a Client Presents a Certificate

In most cases it is important to ensure that clients present valid certificates. However, if you have clients that cannot present a client certificate, or are transitioning to using a certificate authority you may only want to validate certificates from clients that present a certificate.

If you want to bypass validation for clients that don't present certificates, include the sslWeakCertificateValidation (page 1128) run-time option with mongod (page 1049) and mongos (page 1061). If the client does not present a certificate, no validation occurs. These connections, though not validated, are still encrypted using SSL.

For example, consider the following mongod (page 1049) with an SSL configuration that includes the sslWeakCertificateValidation (page 1128) setting:

```
mongod --sslOnNormalPorts --sslWeakCertificateValidation --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCA
```

Then, clients can connect either with the option *--ssl* and **no** certificate or with the option *--ssl* and a **valid** certificate. See *SSL Configuration for Clients* (page 182) for more information on SSL connections for clients.

Note: If the client presents a certificate, the certificate must be a valid certificate.

All connections, including those that have not presented certificates are encrypted using SSL.

Run in FIPS Mode

If your `mongod` (page 1049) or `mongos` (page 1061) is running on a system with an OpenSSL library configured with the FIPS 140-2 module, you can run `mongod` (page 1049) or `mongos` (page 1061) in FIPS mode, with the `sslFIPSMode` (page 1128) setting.

11.4.2 SSL Configuration for Clients

Clients must have support for SSL to work with a `mongod` (page 1049) or a `mongos` (page 1061) instance that has SSL support enabled. The current versions of the Python, Java, Ruby, Node.js, .NET, and C++ drivers have support for SSL, with full support coming in future releases of other drivers.

`mongo` SSL Configuration

For SSL connections, you must use the `mongo` (page 1066) shell built with SSL support or distributed with MongoDB Enterprise. To support SSL, `mongo` (page 1066) has the following settings:

- *--ssl*
- `--sslPEMKeyFile` (page 1127) with the name of the `.pem` file that contains the SSL certificate and key.
- `--sslCAFile` (page 1128) with the name of the `.pem` file that contains the certificate from the Certificate Authority.
- `--sslPEMKeyPassword` (page 1127) option if the client certificate-key file is encrypted.

Connect to MongoDB Instance with SSL Encryption

To connect to a `mongod` (page 1049) or `mongos` (page 1061) instance that requires *only a SSL encryption mode* (page 180), start `mongo` (page 1066) shell with *--ssl*, as in the following:

```
mongo --ssl
```

Connect to MongoDB Instance that Requires Client Certificates

To connect to a `mongod` (page 1049) or `mongos` (page 1061) that requires *CA-signed client certificates* (page 180), start the `mongo` (page 1066) shell with *--ssl* and the `--sslPEMKeyFile` (page 1127) option to specify the signed certificate-key file, as in the following:

```
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Connect to MongoDB Instance that Validates when Presented with a Certificate

To connect to a `mongod` (page 1049) or `mongos` (page 1061) instance that *only requires valid certificates when the client presents a certificate* (page 181), start `mongo` (page 1066) shell either with the `--ssl` ssl and **no** certificate or with the `--ssl` ssl and a **valid** signed certificate.

For example, if `mongod` (page 1049) is running with weak certificate validation, both of the following `mongo` (page 1066) shell clients can connect to that `mongod` (page 1049):

```
mongo --ssl
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Important: If the client presents a certificate, the certificate must be valid.

MMS Monitoring Agent

The Monitoring agent will also have to connect via SSL in order to gather its stats. Because the agent already utilizes SSL for its communications to the MMS servers, this is just a matter of enabling SSL support in MMS itself on a per host basis.

Use the "Edit" host button (i.e. the pencil) on the Hosts page in the MMS console to enable SSL.

Please see the MMS documentation for more information about MMS configuration.

PyMongo

Add the "`ssl=True`" parameter to a PyMongo `MongoClient` to create a MongoDB connection to an SSL MongoDB instance:

```
from pymongo import MongoClient
c = MongoClient(host="mongodb.example.net", port=27017, ssl=True)
```

To connect to a replica set, use the following operation:

```
from pymongo import MongoReplicaSetClient
c = MongoReplicaSetClient("mongodb.example.net:27017",
                          replicaSet="mysetname", ssl=True)
```

PyMongo also supports an "`ssl=true`" option for the MongoDB URI:

```
mongodb://mongodb.example.net:27017/?ssl=true
```

Java

Consider the following example "`SSLApp.java`" class file:

```
import com.mongodb.*;
import javax.net.ssl.SSLSocketFactory;

public class SSLApp {

    public static void main(String args[])  throws Exception {

        MongoClientOptions o = new MongoClientOptions.Builder()
                .socketFactory(SSLSocketFactory.getDefault())
```

```
                .build();

        MongoClient m = new MongoClient("localhost", o);

        DB db = m.getDB( "test" );
        DBCollection c = db.getCollection( "foo" );

        System.out.println( c.findOne() );
    }
}
```

Ruby

The recent versions of the Ruby driver have support for connections to SSL servers. Install the latest version of the driver with the following command:

```
gem install mongo
```

Then connect to a standalone instance, using the following form:

```
require 'rubygems'
require 'mongo'

connection = MongoClient.new('localhost', 27017, :ssl => true)
```

Replace `connection` with the following if you're connecting to a replica set:

```
connection = MongoReplicaSetClient.new(['localhost:27017'],
                                       ['localhost:27018'],
                                       :ssl => true)
```

Here, `mongod` (page 1049) 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` (page 1049) or `mongos` (page 1061) 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` (page 1049) and `mongos` (page 1061) 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);
```

11.5 Monitor MongoDB with SNMP

New in version 2.2.

Enterprise Feature

This feature is only available in MongoDB Enterprise.

This document outlines the use and operation of MongoDB's SNMP extension, which is only available in MongoDB Enterprise.

11.5.1 Prerequisites

Install MongoDB Enterprise

MongoDB Enterprise

Included Files

The Enterprise packages contain the following files:

- `MONGO-MIB.txt`:

  The `MIB` file that describes the data (i.e. schema) for MongoDB's SNMP output

- `mongod.conf`:

  The SNMP configuration file for reading the SNMP output of MongoDB. The SNMP configures the community names, permissions, access controls, etc.

Required Packages

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

- Ubuntu 11.04 requires `libssl0.9.8`, `snmp-mibs-downloader`, `snmp`, and `snmpd`. Issue a command such as the following to install these packages:

  ```
  sudo apt-get install libssl0.9.8 snmp snmpd snmp-mibs-downloader
  ```

- Red Hat Enterprise Linux 6.x series and Amazon Linux AMI require libssl, net-snmp, net-snmp-libs, and net-snmp-utils. Issue a command such as the following to install these packages:

  ```
  sudo yum install libssl net-snmp net-snmp-libs net-snmp-utils
  ```

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

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

11.5.2 Configure SNMP

Install MIB Configuration Files

Ensure that the MIB directory http://docs.mongodb.org/manualusr/share/snmp/mibs exists. If not, issue the following command:

```
sudo mkdir -p /usr/share/snmp/mibs
```

Use the following command to create a symbolic link:

```
sudo ln -s <path>MONGO-MIB.txt /usr/share/snmp/mibs/
```

Replace [/path/to/mongodb/distribution/] with the path to your MONGO-MIB.txt configuration file.

Copy the mongod.conf file into the /etc/snmp directory with the following command:

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

Start Up

You can control MongoDB Enterprise using default or custom control scripts, just as with any other **mongod**:

Use the following command to view all SNMP options available in your MongoDB:

```
mongod --help | grep snmp
```

The above command should return the following output:

```
Module snmp options:
  --snmp-subagent       run snmp subagent
  --snmp-master         run snmp as master
```

Ensure that the following directories exist:

- http://docs.mongodb.org/manualdata/db/ (This is the path where MongoDB stores the data files.)
- http://docs.mongodb.org/manualvar/log/mongodb/ (This is the path where MongoDB writes the log output.)

If they do not, issue the following command:

```
mkdir -p /var/log/mongodb/ /data/db/
```

Start the **mongod** instance with the following command:

```
mongod --snmp-master --port 3001 --fork --dbpath /data/db/  --logpath /var/log/mongodb/1.log
```

Optionally, you can set these options in a *configuration file* (page 1115).

To check if **mongod** is running with SNMP support, issue the following command:

```
ps -ef | grep 'mongod --snmp'
```

The command should return output that includes the following line. This indicates that the proper **mongod** instance is running:

```
systemuser 31415 10260  0 Jul13 pts/16   00:00:00 mongod --snmp-master --port 3001 # [...]
```

Test SNMP

Check for the snmp agent process listening on port `1161` with the following command:

```
sudo lsof -i :1161
```

which return the following output:

```
COMMAND   PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
mongod  9238 sysadmin   10u  IPv4  96469      0t0  UDP localhost:health-polling
```

Similarly, this command:

```
netstat -anp | grep 1161
```

should return the following output:

```
udp      0      0 127.0.0.1:1161      0.0.0.0:*      9238/<path>/mongod
```

Run `snmpwalk` Locally

`snmpwalk` provides tools for retrieving and parsing the SNMP data according to the MIB. If you installed all of the required packages above, your system will have `snmpwalk`.

Issue the following command to collect data from **mongod** using SNMP:

```
snmpwalk -m MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

You may also choose to specify the path to the MIB file:

```
snmpwalk -m /usr/share/snmp/mibs/MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

Use this command *only* to ensure that you can retrieve and validate SNMP data from MongoDB.

11.5.3 Troubleshooting

Always check the logs for errors if something does not run as expected; see the log at `http://docs.mongodb.org/manualvar/log/mongodb/1.log`. The presence of the following line indicates that the **mongod** cannot read the `/etc/snmp/mongod.conf` file:

```
[SNMPAgent] warning: error starting SNMPAgent as master err:1
```

11.6 Manage `mongod` Processes

MongoDB runs as a standard program. You can start MongoDB from a command line by issuing the `mongod` (page 1049) command and specifying options. For a list of options, see *mongod* (page 1049). MongoDB can also run as a Windows service. For details, see *MongoDB as a Windows Service* (page 18). To install MongoDB, see *Install MongoDB* (page 3).

The following examples assume the directory containing the `mongod` (page 1049) process is in your system paths. The `mongod` (page 1049) process is the primary database process that runs on an individual server. `mongos` (page 1061) provides a coherent MongoDB interface equivalent to a `mongod` (page 1049) from the perspective of a client. The `mongo` (page 1066) binary provides the administrative shell.

This document page discusses the `mongod` (page 1049) process; however, some portions of this document may be applicable to `mongos` (page 1061) instances.

See also:

Run-time Database Configuration (page 129), *mongod* (page 1049), *mongos* (page 1061), and *Configuration File Options* (page 1115).

11.6.1 Start `mongod`

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

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

```
mongod
```

Specify a Data Directory

If you want `mongod` (page 1049) to store data files at a path *other than* `http://docs.mongodb.org/manualdata/db` you can specify a `dbpath` (page 1118). The `dbpath` (page 1118) must exist before you start `mongod` (page 1049). If it does not exist, create the directory and the permissions so that `mongod` (page 1049) can read and write data to this path. For more information on permissions, see the *security operations documentation* (page 210).

To specify a `dbpath` (page 1118) for `mongod` (page 1049) to use as a data directory, use the `--dbpath` option. The following invocation will start a `mongod` (page 1049) instance and store data in the `http://docs.mongodb.org/manualsrv/mongodb` path

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

Specify a TCP Port

Only a single process can listen for connections on a network interface at a time. If you run multiple `mongod` (page 1049) processes on a single machine, or have other processes that must use this port, you must assign each a different port to listen on for client connections.

To specify a port to `mongod` (page 1049), use the `--port` option on the command line. The following command starts `mongod` (page 1049) listening on port `12345`:

```
mongod --port 12345
```

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

Start `mongod` as a Daemon

To run a `mongod` (page 1049) process as a daemon (i.e. `fork` (page 1117),) *and* write its output to a log file, use the `--fork` and `--logpath` options. You must create the log directory; however, `mongod` (page 1049) will create the log file if it does not exist.

The following command starts `mongod` (page 1049) as a daemon and records log output to `http://docs.mongodb.org/manualvar/log/mongodb.log`.

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

Additional Configuration Options

For an overview of common configurations and common configuration deployments. configurations for common use cases, see *Run-time Database Configuration* (page 129).

11.6.2 Stop `mongod`

To stop a `mongod` (page 1049) instance not running as a daemon, press `Control+C`. MongoDB stops when all ongoing operations are complete and does a clean exit, flushing and closing all data files.

To stop a `mongod` (page 1049) instance running in the background or foreground, issue the `shutdownServer()` (page 1013) helper in the `mongo` (page 1066) shell. Use the following sequence:

1. To open the `mongo` (page 1066) shell for a `mongod` (page 1049) instance running on the default port of `27017`, issue the following command:

   ```
   mongo
   ```

2. To switch to the `admin` database and shutdown the `mongod` (page 1049) instance, issue the following commands:

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

You may only use `db.shutdownServer()` (page 1013) when connected to the `mongod` (page 1049) when authenticated to the `admin` database or on systems without authentication connected via the localhost interface.

Alternately, you can shut down the `mongod` (page 1049) instance:

- using the `--shutdown` option
- from a driver using the `shutdown` (page 896). For details, see the *drivers documentation* (page 575) for your driver.

`mongod` Shutdown and Replica Sets

If the `mongod` (page 1049) is the *primary* in a *replica set*, the shutdown process for these `mongod` (page 1049) instances has the following steps:

1. Check how up-to-date the *secondaries* are.
2. If no secondary is within 10 seconds of the primary, `mongod` (page 1049) will return a message that it will not shut down. You can pass the `shutdown` (page 896) command a `timeoutSecs` argument to wait for a secondary to catch up.
3. If there is a secondary within 10 seconds of the primary, the primary will step down and wait for the secondary to catch up.

4. After 60 seconds or once the secondary has caught up, the primary will shut down.

If there is no up-to-date secondary and you want the primary to shut down, issue the shutdown (page 896) command with the force argument, as in the following mongo (page 1066) shell operation:

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

To keep checking the secondaries for a specified number of seconds if none are immediately up-to-date, issue shutdown (page 896) with the timeoutSecs argument. MongoDB will keep checking the secondaries for the specified number of seconds if none are immediately up-to-date. If any of the secondaries catch up within the allotted time, the primary will shut down. If no secondaries catch up, it will not shut down.

The following command issues shutdown (page 896) with timeoutSecs set to 5:

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

Alternately you can use the timeoutSecs argument with the shutdownServer() (page 1013) method:

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

11.6.3 Sending a UNIX INT or TERM Signal

You can cleanly stop mongod (page 1049) using a SIGINT or SIGTERM signal on UNIX-like systems. Either ^C for a non-daemon mongod (page 1049) instance, kill -2 <pid>, or kill -15 <pid> will cleanly terminate the mongod (page 1049) instance.

Terminating a mongod (page 1049) instance that is **not** running with *journaling* with kill -9 <pid> (i.e. SIGKILL) will probably cause data corruption.

To recover data in situations where mongod (page 1049) instances have not terminated cleanly *without journaling* see *Recover MongoDB Data following Unexpected Shutdown* (page 144).

11.7 Rotate Log Files

11.7.1 Overview

Log rotation archives the current log file and starts a new one. Specifically, log rotation renames the current log file by appending the filename with a timestamp, [1] opens a new log file, and finally closes the old log. MongoDB will only rotate logs, when you use the logRotate (page 897) command, or issue the process a SIGUSR1 signal as described in this procedure.

See also:

For information on logging, see the *Process Logging* (page 161) section.

11.7.2 Procedure

The following steps create and rotate a log file:

1. Start a mongod (page 1049) with verbose logging, with appending enabled, and with the following log file:

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

2. In a separate terminal, list the matching files:

[1] MongoDB renders this timestamp in UTC (GMT) and formatted as *ISODate*.

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

For results, you get:

```
server1.log
```

3. Rotate the log file using *one* of the following methods.

   - From the `mongo` (page 1066) shell, issue the `logRotate` (page 897) command from the `admin` database:

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

     This is the only available method to rotate log files on Windows systems.

   - From the UNIX shell, rotate logs for a single process by issuing the following command:

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

   - From the UNIX shell, rotate logs for all `mongod` (page 1049) processes on a machine by issuing the following command:

     ```
     killall -SIGUSR1 mongod
     ```

4. List the matching files again:

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

   For results you get something similar to the following. The timestamps will be different.

   ```
   server1.log  server1.log.2011-11-24T23-30-00
   ```

   The example results indicate a log rotation performed at exactly `11:30 pm` on `November 24th, 2011 UTC`, which is the local time offset by the local time zone. The original log file is the one with the timestamp. The new log is `server1.log` file.

   If you issue a second `logRotate` (page 897) command an hour later, then an additional file would appear when listing matching files, as in the following example:

   ```
   server1.log  server1.log.2011-11-24T23-30-00  server1.log.2011-11-25T00-30-00
   ```

   This operation does not modify the `server1.log.2011-11-24T23-30-00` file created earlier, while `server1.log.2011-11-25T00-30-00` is the previous `server1.log` file, renamed. `server1.log` is a new, empty file that receives all new log output.

CHAPTER 12

Reference

12.1 UNIX `ulimit` Settings

Most UNIX-like operating systems, including Linux and OS X, provide ways to limit and control the usage of system resources such as threads, files, and network connections on a per-process and per-user basis. These limits prevent single users from using too many system resources. Sometimes, these limits have low default values that can cause a number of issues in the course of normal MongoDB operation.

12.1.1 Resource Utilization

`mongod` (page 1049) and `mongos` (page 1061) 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` (page 1049) and `mongos` (page 1061) 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` (page 1049) instance.
- 1 file descriptor for each journal file used by the `mongod` (page 1049) instance when `journal` (page 1119) is `true`.
- In replica sets, each `mongod` (page 1049) maintains a connection to all other members of the set.

`mongod` (page 1049) uses background threads for a number of internal processes, including *TTL collections* (page 599), 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 1061) must maintain connects to all config servers and all shards, which includes all members of all replica sets.

For `mongos` (page 1061), consider the following behaviors:

- `mongos` (page 1061) instances maintain a connection pool to each shard so that the `mongos` (page 1061) 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 1116) run-time option.

  By restricting the number of incoming connections you can prevent a cascade effect where the `mongos` (page 1061) creates too many connections on the `mongod` (page 1049) instances.

  Note: You cannot set `maxConns` (page 1116) to a value higher than *20000*.

12.1.2 Review and Set Resource Limits

`ulimit`

You can use the `ulimit` command at the system prompt to check system limits, as in the following example:

```
$ ulimit -a
-t: cpu time (seconds)         unlimited
-f: file size (blocks)         unlimited
-d: data seg size (kbytes)     unlimited
-s: stack size (kbytes)        8192
-c: core file size (blocks)    0
-m: resident set size (kbytes) unlimited
-u: processes                  192276
-n: file descriptors           21000
-l: locked-in-memory size (kb) 40000
-v: address space (kb)         unlimited
-x: file locks                 unlimited
-i: pending signals            192276
-q: bytes in POSIX msg queues  819200
-e: max nice                   30
-r: max rt priority            65
-N 15:                         unlimited
```

`ulimit` refers to the per-*user* limitations for various resources. Therefore, if your `mongod` (page 1049) instance executes as a user that is also running multiple processes, or multiple `mongod` (page 1049) processes, you might see contention for these resources. Also, be aware that the `processes` value (i.e. `-u`) refers to the combined number of distinct processes and sub-process threads.

You can change `ulimit` settings by issuing a command in the following form:

```
ulimit -n <value>
```

For many distributions of Linux you can change values by substituting the `-n` option for any possible value in the output of `ulimit -a`. On OS X, use the `launchctl limit` command. See your operating system documentation for the precise procedure for changing system limits on running systems.

Note: After changing the `ulimit` settings, you *must* restart the process to take advantage of the modified settings. You can use the `http://docs.mongodb.org/manualproc` file system to see the current limitations on a running process.

Depending on your system's configuration, and default settings, any change to system limits made using `ulimit` may revert following system a system restart. Check your distribution and operating system documentation for more information.

`/proc` File System

Note: This section applies only to Linux operating systems.

The http://docs.mongodb.org/manualproc file-system stores the per-process limits in the file system object located at /proc/<pid>/limits, where <pid> is the process's *PID* or process identifier. You can use the following bash function to return the content of the limits object for a process or processes with a given name:

```
return-limits(){

     for process in $@; do
          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
```

12.1.3 Recommended Settings

Every deployment may have unique requirements and settings; however, the following thresholds and settings are particularly important for mongod (page 1049) and mongos (page 1061) deployments:

- -f (file size): unlimited

- `-t` (cpu time): `unlimited`
- `-v` (virtual memory): `unlimited` [1]
- `-n` (open files): `64000`
- `-m` (memory size): `unlimited` [1]
- `-u` (processes/threads): `32000`

Always remember to restart your `mongod` (page 1049) and `mongos` (page 1061) instances after changing the `ulimit` settings to make sure that the settings change takes effect.

12.2 Production Notes

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

12.2.1 Backups

To make backups of your MongoDB database, please refer to *Backup Strategies for MongoDB Systems* (page 133).

12.2.2 Networking

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

See documents in the *Security* (page 205) section for additional information, specifically:

- *Interfaces and Port Numbers* (page 208)
- *Firewalls* (page 209)
- *Configure Linux iptables Firewall for MongoDB* (page 215)
- *Configure Windows netsh Firewall for MongoDB* (page 219)

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

12.2.3 MongoDB on Linux

If you use the Linux kernel, the MongoDB user community has recommended Linux kernel 2.6.36 or later for running MongoDB in production.

Because MongoDB preallocates its database files before using them and because MongoDB uses very large files on average, you should use the Ext4 and XFS file systems if using the Linux kernel:

- If you use the Ext4 file system, use at least version 2.6.23 of the Linux Kernel.
- If you use the XFS file system, use at least version 2.6.25 of the Linux Kernel.
- If you are using a Red Hat 5.6, and wish to use ext4, use at least version `2.6.18-245.el5` of the Linux Kernel.

[1] If you limit virtual or resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

For MongoDB on Linux use the following recommended configurations:

- Turn off `atime` for the storage volume with the *database files*.
- Set the file descriptor limit and the user process limit above 20,000, according to the suggestions in *UNIX ulimit Settings* (page 193). A low ulimit will affect MongoDB when under heavy use and will produce weird errors.
- Do not use `hugepages` virtual memory pages, MongoDB performs better with normal virtual memory pages.
- Disable NUMA in your BIOS. If that is not possible see *NUMA* (page 198).
- Ensure that readahead settings for the block devices that store the database files are acceptable. See the *Readahead* (page 197) section
- Use NTP to synchronize time among your hosts. This is especially important in sharded clusters.

12.2.4 Readahead

For random access use patterns set readahead values low, for example setting readahead to a small value such as 32 (16KB) often works well.

12.2.5 MongoDB on Virtual Environments

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

EC2

MongoDB is compatible with EC2 and requires no configuration changes specific to the environment.

VMWare

MongoDB is compatible with VMWare. Some in the MongoDB community have run into issues with VMWare's memory overcommit feature and suggest disabling the feature.

You can clone a virtual machine running MongoDB. You might use this to spin up a new virtual host that will be added as a member of a replica set. If journaling is enabled, the clone snapshot will be consistent. If not using journaling, stop `mongod` (page 1049), clone, and then restart.

OpenVZ

The MongoDB community has encountered issues running MongoDB on OpenVZ.

12.2.6 Disk and Storage Systems

Swap

Configure swap space for your systems. Having swap can prevent issues with memory contention and can prevent the OOM Killer on Linux systems from killing `mongod` (page 1049). Because of the way `mongod` (page 1049) maps memory files to memory, the operating system will never store MongoDB data in swap.

RAID

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

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

RAID-0 provides good write performance but provides limited availability, and reduced performance on read operations, particularly using Amazon's EBS volumes: as a result, avoid RAID-0 with MongoDB deployments.

Remote Filesystems

Some versions of NFS perform very poorly with MongoDB and NFS is not recommended for use with MongoDB. Performance problems arise when both the data files and the journal files are both hosted on NFS: you may experience better performance if you place the journal on local or `iscsi` volumes. If you must use NFS, add the following NFS options to your `/etc/fstab` file: `bg`, `nolock`, and `noatime`.

Many MongoDB deployments work successfully with Amazon's *Elastic Block Store* (EBS) volumes. There are certain intrinsic performance characteristics, with EBS volumes that users should consider.

12.2.7 Hardware Requirements and Limitations

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

When installing hardware for MongoDB, consider the following:

- As with all software, more RAM and a faster CPU clock speed are important to productivity.
- Because databases do not perform high amounts of computation, increasing the number cores helps but does not provide a high level of marginal return.
- MongoDB has good results and good price/performance with SATA SSD (Solid State Disk) and with PCI (Peripheral Component Interconnect).
- Commodity (SATA) spinning drives are often a good option as the speed increase for random I/O for more expensive drives is not that dramatic (only on the order of 2x), spending that money on SSDs or RAM may be more effective.

MongoDB on NUMA Hardware

Important: The discussion of NUMA in this section only applies to Linux, and does not apply to deployments where `mongod` (page 1049) instances run other UNIX-like systems or on Windows.

MongoDB and NUMA, Non-Uniform Access Memory, do not work well together. When running MongoDB on NUMA hardware, disable NUMA for MongoDB and run with an interleave memory policy. NUMA can cause a number of operational problems with MongoDB, including slow performance for periods of time or high system processor usage.

Note: On Linux, MongoDB version 2.0 and greater checks these settings on start up and prints a warning if the system is NUMA-based.

To disable NUMA for MongoDB, use the `numactl` command and start `mongod` (page 1049) in the following manner:

```
numactl --interleave=all /usr/bin/local/mongod
```

Adjust the `proc` settings using the following command:

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

To fully disable NUMA you must perform both operations. However, you can change `zone_reclaim_mode` without restarting mongod. For more information, see documentation on Proc/sys/vm.

See the The MySQL "swap insanity" problem and the effects of NUMA post, which describes the effects of NUMA on databases. This blog post addresses the impact of NUMA for MySQL; however, the issues for MongoDB are similar. The post introduces NUMA its goals, and illustrates how these goals are not compatible with production databases.

12.2.8 Performance Monitoring

iostat

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

For example:

```
iostat -xmt 1
```

Use the mount command to see what device your `data directory` (page 1118) resides on.

Key fields from `iostat`:

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

bwm-ng

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

12.2.9 Production Checklist

64-bit Builds for Production

Always use 64-bit Builds for Production. MongoDB uses memory mapped files. See the *32-bit limitations* (page 734) for more information.

32-bit builds exist to support use on development machines and also for other miscellaneous things such as replica set arbiters.

BSON Document Size Limit

There is a `BSON Document Size` (page 1139) limit – at the time of this writing 16MB per document. If you have large objects, use *GridFS* (page 70) instead.

Set Appropriate Write Concern for Write Operations

See *Write Concern* (page 395) for more information.

Dynamic Schema

Data in MongoDB has a *dynamic schema*. *Collections* do not enforce *document* structure. This facilitates iterative development and polymorphism. However, collections often hold documents with highly homogeneous structures. See *Data Modeling Considerations for MongoDB Applications* (page 107) for more information.

Some operational considerations include:

- the exact set of collections to be used
- the indexes to be used, which are created explicitly except for the `_id` index
- shard key declarations, which are explicit and quite important as it is hard to change shard keys later

One very simple rule-of-thumb is not to import data from a relational database unmodified: you will generally want to "roll up" certain data into richer documents that use some embedding of nested documents and arrays (and/or arrays of subdocuments).

Updates by Default Affect Only one Document

Set the `multi` parameter to `true` to `update()` (page 974) multiple documents that meet the query criteria. The `mongo` (page 1066) shell syntax is:

```
db.records.update(my_query, my_update_expression, bool_upsert, bool_multi)
```

Set `bool_multi` to `true` when updating many documents. Otherwise only the first matched will update.

Case Sensitive Strings

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

Consider:

- storing data in a normalized case format, or
- using regular expressions ending with `http://docs.mongodb.org/manuali`
- and/or using *$toLower* (page 283) or *$toUpper* (page 284) in the *aggregation framework* (page 247)

Type Sensitive Fields

MongoDB data – which is JSON-style, specifically, BSON format – have several data types.

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

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

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

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

Locking

Older versions of MongoDB used a "global lock"; use MongoDB v2.2+ for better results. See the *Concurrency* (page 749) page for more information.

Packages

Be sure you have the latest stable release if you are using a package manager. You can see what is current on the Downloads page, even if you then choose to install via a package manager.

Use Odd Number of Replica Set Members

Replica sets (page 367) perform consensus elections. Use either an odd number of members (e.g., three) or else use an arbiter to get up to an odd number of votes.

Don't disable journaling

See *Journaling* (page 155) for more information.

Keep Replica Set Members Up-to-Date

This is important as MongoDB replica sets support automatic failover. Thus you want your secondaries to be up-to-date. You have a few options here:

1. Monitoring and alerts for any lagging can be done via various means. The MongoDB Management Service (MMS) shows a graph of replica set lag
2. Using *getLastError* (page 395) with `w:'majority'`, you will get a timeout or no return if a majority of the set is lagging. This is thus another way to guard against lag and get some reporting back of its occurrence.
3. Or, if you want to fail over manually, you can set your secondaries to `priority:0` in their configurations. Then manual action would be required for a failover. This is practical for a small cluster; for a large cluster you will want automation.

Additionally, see information on *replica set rollbacks* (page 393).

Additional Deployment Considerations

- Pick your shard keys carefully! There is no way to modify a shard key on a collection that is already sharded.
- You cannot shard an existing collection over 256 gigabytes. To shard large amounts of data, create a new empty sharded collection, and ingest the data from the source collection using an application level import operation.
- Unique indexes are not enforced across shards except for the shard key itself. See *Enforce Unique Keys for Sharded Collections* (page 548).
- Consider *pre-splitting* (page 515) a sharded collection before a massive bulk import. Usually this isn't necessary but on a bulk import of size it is helpful.
- Use *security/auth* (page 207) mode if you need it. By default `auth` (page 1118) is not enabled and `mongod` (page 1049) assumes a trusted environment.
- You do not have *fully generalized transactions* (page 593). Create rich documents and read the preceding link and consider the use case – often there is a good fit.

- Disable NUMA for best results. If you have NUMA enabled, `mongod` (page 1049) will print a warning when it starts.
- Avoid excessive prefetch/readahead on the filesystem. Check your prefetch settings. Note on linux the parameter is in *sectors*, not bytes. 32KBytes (a setting of 64 sectors) is pretty reasonable.
- Check *ulimit* (page 193) settings.
- Use SSD if available and economical. Spinning disks can work well but SSDs' capacity for random I/O operations work well with the update model of `mongod` (page 1049). See *Remote Filesystems* (page 198) for more info.
- Ensure that clients keep reasonable pool sizes to avoid overloading the connection tracking capacity of a single `mongod` (page 1049) or `mongos` (page 1061) instance.

See also:

- *Sharded Cluster Tutorials* (page 515)
- *Replica Set Tutorials* (page 415)
- *Indexing Operations* (page 329)

Part V

Security

The documentation in this section outlines basic security, risk management, and access control, and includes specific tasks for configuring firewalls, authentication, and system privileges. User roles in MongoDB provide granular control over user authorization and access.

If you believe you have discovered a vulnerability in MongoDB, please see *Create a Vulnerability Report* (page 222).

CHAPTER 13

Security Concepts and Strategies

13.1 Security Practices and Management

This document describes risk mitigation in MongoDB deployments. 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 a *Defense In Depth* approach is to ensure there are no exploitable points of failure in your deployment that could allow an intruder or un-trusted party to access the data stored in the MongoDB database. The easiest and most effective way to reduce the risk of exploitation is to run MongoDB in a trusted environment, limit access, follow a system of least privilege, and follow best development and deployment practices. See the *Strategies for Reducing Risk* (page 207) section.

For an outline of all security, authentication, and authorization documentation, see *Security* (page 205).

13.1.1 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` (page 1049), `mongos` (page 1061) 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` (page 1049) and `mongos` (page 1061) 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* (page 211) for access to MongoDB instances.
- requiring strong, complex, single purpose authentication credentials. This should be part of your internal security policy.
- 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.

13.1.2 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 *Create a Vulnerability Report* (page 222) document.

13.1.3 Runtime Security Configuration

For configuration settings that affect security, see *Security Considerations* (page 130).

13.1.4 Networking Risk Exposure

Interfaces and Port Numbers

The following list includes all default ports used by MongoDB:

27017 This is the default port for `mongod` (page 1049) and `mongos` (page 1061) instances. You can change this port with `port` (page 1116) or *--port*.

27018 This is the default port when running with *--shardsvr* runtime operation or `shardsvr` (page 1125) setting.

27019 This is the default port when running with *--configsvr* runtime operation or `configsvr` (page 1125) 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 1116).

By default MongoDB programs (i.e. `mongos` (page 1061) and `mongod` (page 1049)) 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.

Network Interface Limitation

You can limit the network exposure with the following configuration options:

- the `nohttpinterface` (page 1120) setting for `mongod` (page 1049) and `mongos` (page 1061) instances.

  Disables the "home" status page, which would run on port `28017` by default. The status interface is read-only by default. You may also specify this option on the command line as *mongod --nohttpinterface* or *mongos --nohttpinterface*. Authentication does not control or affect access to this interface.

  Important: Disable this option for production deployments. If you *do* leave this interface enabled, you should only allow trusted clients to access this port. See *Firewalls* (page 209).

- the `port` (page 1116) setting for `mongod` (page 1049) and `mongos` (page 1061) instances.

  Changes the main port on which the `mongod` (page 1049) or `mongos` (page 1061) instance listens for connections. Changing the port does not meaningfully reduce risk or limit exposure.

  You may also specify this option on the command line as *mongod --port* or *mongos --port*.

  Whatever port you attach `mongod` (page 1049) and `mongos` (page 1061) instances to, you should only allow trusted clients to connect to this port. See *Firewalls* (page 209).

- the `rest` (page 1121) setting for `mongod` (page 1049).

  Enables a fully interactive administrative *REST* interface, which is *disabled by default*. The status interface, which *is* enabled by default, is read-only. This configuration makes that interface fully interactive. The REST interface does not support any authentication and you should always restrict access to this interface to only allow trusted clients to connect to this port.

  You may also enable this interface on the command line as *mongod --rest*.

  Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

- the `bind_ip` (page 1116) setting for `mongod` (page 1049) and `mongos` (page 1061) 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 1116) a comma separated list of IP addresses. You can use the *mongod --bind_ip* and *mongos --bind_ip* option on the command line at run time to limit the network accessibility of a MongoDB program.

  Important: Make sure that your `mongod` (page 1049) and `mongos` (page 1061) 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.

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` (page 1049) and `mongos` (page 1061) instances and that the `mongod` (page 1049) and `mongos` (page 1061) instances can only connect to trusted outputs.

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2 page, which addresses Amazon's Security Groups and other EC2-specific security features.

Virtual Private Networks

Virtual private networks, or VPNs, make it possible to link two networks over an encrypted and limited-access trusted network. Typically MongoDB users who use VPNs use SSL rather than IPSEC VPNs for performance issues.

Depending on configuration and implementation VPNs provide for certificate validation and a choice of encryption protocols, which requires a rigorous level of authentication and identification of all clients. Furthermore, because VPNs provide a secure tunnel, using a VPN connection to control access to your MongoDB instance, you can prevent tampering and "man-in-the-middle" attacks.

13.1.5 Operations

Always run the `mongod` (page 1049) or `mongos` (page 1061) 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` (page 1049) or `mongos` (page 1061) 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 MongoDB authentication. See *Access Control* (page 211).

13.1.6 Interfaces

Simply limiting access to a `mongod` (page 1049) is not sufficient for totally controlling risk exposure. Consider the recommendations in the following section, for limiting exposure other interface-related risks.

JavaScript and the Security of the `mongo` Shell

Be aware of the following capabilities and behaviors of the `mongo` (page 1066) shell:

- `mongo` (page 1066) will evaluate a `.js` file passed to the *`mongo --eval`* option. The `mongo` (page 1066) shell does not validate the input of JavaScript input to *`--eval`*.
- `mongo` (page 1066) 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` (page 1066) 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`.

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` (page 1049) or `mongos` (page 1061) instance. The HTTP interface is always available on the port numbered `1000` greater than the primary `mongod` (page 1049) port. By default this is `28017`, but is indirectly set using the `port` (page 1116) option which allows you to configure the primary `mongod` (page 1049) port.

Without the `rest` (page 1121) 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 1120) run time option or the *`--nohttpinterface`* command line option.

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 when running with `auth` (page 1118) 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)

13.1.7 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 HIPAA, PCI-DSS, and FERPA. For more information consider the following resources:

- Datasheet
- Webinar

13.2 Access Control

MongoDB provides support for authentication and authorization by storing a user's credentials and privileges in a database's `system.users` (page 238) collection. MongoDB provisions authentication and access on a per-database level. Users exist in the context of a single logical database.

For MongoDB Enterprise installations, MongoDB also provides support for authentication using a Kerberos service. See *Deploy MongoDB with Kerberos Authentication* (page 228).

13.2.1 Authentication

MongoDB provides support for basic authentication by:

- storing user credentials in a database's `system.users` (page 238) collection, and
- providing the `auth` (page 1118) and `keyFile` (page 1117) configuration settings to enable authentication for a given `mongod` (page 1049) or `mongos` (page 1061) instance.

Authentication is **disabled** by default.

To enable authentication, see the following:

- *Enable Authentication* (page 224)
- *Deploy MongoDB with Kerberos Authentication* (page 228)

13.2.2 Authorization

MongoDB supports role-based access to databases and database operations by storing each user's roles in a *privilege document* (page 233) in the `system.users` (page 238) collection. For a description of privilege documents and of available roles, see *User Privilege Roles in MongoDB* (page 233).

Changed in version 2.4: The schema of `system.users` (page 238) changed to accommodate a more sophisticated user privilege model, as defined in *privilege documents* (page 233).

The `system.users` (page 238) collection is protected to prevent privilege escalation attacks. To access the collection, you must have the `userAdmin` (page 235) or `userAdminAnyDatabase` (page 237) role.

To assign user roles, you must first create an admin user in the database. Then you create additional users, assigning them appropriate user roles.

To assign user roles, see the following:

- *Create a User Administrator* (page 224)
- *Add a User to a Database* (page 226)

User Roles in the admin Database

The `admin` database provides roles not available in other databases, including a role that effectively makes a user a MongoDB system superuser. See *Database Administration Roles* (page 234) and *Administrative Roles* (page 235).

Authentication to One Database at a Time

You can log in as only one user for a given database, including the `admin` database. If you authenticate to a database as one user and later authenticate on the same database as a different user, the second authentication invalidates the first. Logging into a *different* database, however, does not invalidate authentication on other databases.

13.3 Inter-Process Authentication

In most cases, *replica set* and *sharded cluster* administrators do not have to keep additional considerations in mind beyond the normal security precautions that all MongoDB administrators must take. However, ensure that:

- Your network configuration will allow every member of the replica set to contact every other member of the replica set.
- If you use MongoDB's authentication system to limit access to your infrastructure, ensure that you configure a `keyFile` (page 1117) on all members to permit authentication.

For most instances, the most effective ways to control access and to secure the connection between members of a *replica set* depend on network-level access control. Use your environment's firewall and network routing to ensure that traffic *only* from clients and other replica set members can reach your `mongod` (page 1049) 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` (page 1049) and `mongos` (page 1061) 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: Added support authentication in replica set deployments.

Changed in version 1.9.1: Added support authentication in sharded replica set deployments.

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* (or *mongos --keyFile*) options on the command line.

Setting `keyFile` (page 1117) 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 1061) 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. See *Generate a Key File* (page 227) for instructions on generating a key file.

CHAPTER 14

Tutorials

14.1 Network Security

14.1.1 Configure Linux `iptables` Firewall for MongoDB

On contemporary Linux systems, the `iptables` program provides methods for managing the Linux Kernel's `netfilter` or network packet filtering capabilities. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic firewall configurations for `iptables` firewalls on Linux. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see *Security Practices and Management* (page 207).

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2 page, which addresses Amazon's Security Groups and other EC2-specific security features.

Overview

Rules in `iptables` configurations fall into chains, which describe the process for filtering and processing specific streams of traffic. Chains have an order, and packets must pass through earlier rules in a chain to reach later rules. This document addresses only the following two chains:

`INPUT` Controls all incoming traffic.

`OUTPUT` Controls all outgoing traffic.

Given the *default ports* (page 208) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod` (page 1049) and `mongos` (page 1061) instances.

Be aware that, by default, the default policy of `iptables` is to allow all connections and traffic unless explicitly disabled. The configuration changes outlined in this document will create rules that explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed. When you have properly configured your `iptables` rules to allow only the traffic that you want to permit, you can *Change Default Policy to DROP* (page 218).

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 1116) configuration setting, you will need to modify the rules accordingly.

Traffic to and from `mongod` Instances

This pattern is applicable to all `mongod` (page 1049) 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` (page 1049) instance from the application server. In the following examples, replace `<ip-address>` with the IP address of the application server:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27017 -m state --state NEW,ESTABLISHED -j
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` (page 1049) instance. The second rule, allows outgoing traffic from the `mongod` (page 1049) 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/manual24` using one of the following specifications for the `<ip-address>`, as follows:

```
10.10.10.10/24
10.10.10.10/255.255.255.0
```

Traffic to and from `mongos` Instances

`mongos` (page 1061) instances provide query routing for *sharded clusters*. Clients connect to `mongos` (page 1061) instances, which behave from the client's perspective as `mongod` (page 1049) instances. In turn, the `mongos` (page 1061) connects to all `mongod` (page 1049) 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` (page 1049) 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.

Traffic to and from a MongoDB Config Server

Config servers, host the *config database* that stores metadata for sharded clusters. Each production cluster has three config servers, initiated using the *`mongod --configsvr`* option. [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 -j
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT
```

[1] You can also run a config server by setting the `configsvr` (page 1125) option in a configuration file.

Replace <ip-address> with the address or address space of *all* the mongod (page 1049) that provide config servers.

Additionally, config servers need to allow incoming connections from all of the mongos (page 1061) instances in the cluster *and* all mongod (page 1049) instances in the cluster. Add rules that resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -j
```

Replace <ip-address> with the address of the mongos (page 1061) instances and the shard mongod (page 1049) instances.

Traffic to and from a MongoDB Shard Server

For shard servers, running as *mongod --shardsvr* [2] Because the default port number when running with shardsvr (page 1125) is 27018, you must configure the following iptables rules to allow traffic to and from each shard:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27018 -m state --state NEW,ESTABLISHED -j
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 (page 1049). This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members, to:

- all mongod (page 1049) instances in the shard's replica sets.
- all mongod (page 1049) instances in other shards. [3]

Furthermore, shards need to be able make outgoing connections to:

- all mongos (page 1061) instances.
- all mongod (page 1049) 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 1061) instances:

```
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

Provide Access For Monitoring Systems

1. The mongostat (page 1098) diagnostic tool, when running with the *--discover* needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos (page 1061) 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 (page 1049) instances running with shardsvr (page 1125), the rule would resemble the following:

[2] You can also specify the shard server option using the shardsvr (page 1125) 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.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28018 -m state --state NEW,ESTABLISH
```

For config server mongod (page 1049) instances running with configsvr (page 1125), the rule would resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28019 -m state --state NEW,ESTABLISH
```

Change Default Policy to DROP

The default policy for `iptables` chains is to allow all traffic. After completing all `iptables` configuration changes, you *must* change the default policy to `DROP` so that all traffic that isn't explicitly allowed as above will not be able to reach components of the MongoDB deployment. Issue the following commands to change this policy:

```
iptables -P INPUT DROP

iptables -P OUTPUT DROP
```

Manage and Maintain iptables Configuration

This section contains a number of basic operations for managing and using `iptables`. There are various front end tools that automate some aspects of `iptables` configuration, but at the core all `iptables` front ends provide the same basic functionality:

Make all iptables Rules Persistent

By default all `iptables` rules are only stored in memory. When your system restarts, your firewall rules will revert to their defaults. When you have tested a rule set and have guaranteed that it effectively controls traffic you can use the following operations to you should make the rule set persistent.

On Red Hat Enterprise Linux, Fedora Linux, and related distributions you can issue the following command:

```
service iptables save
```

On Debian, Ubuntu, and related distributions, you can use the following command to dump the `iptables` rules to the `/etc/iptables.conf` file:

```
iptables-save > /etc/iptables.conf
```

Run the following operation to restore the network rules:

```
iptables-restore < /etc/iptables.conf
```

Place this command in your `rc.local` file, or in the `/etc/network/if-up.d/iptables` file with other similar operations.q

List all iptables Rules

To list all of currently applied `iptables` rules, use the following operation at the system shell.

```
iptables --L
```

Flush all `iptables` Rules

If you make a configuration mistake when entering `iptables` rules or simply need to revert to the default rule set, you can use the following operation at the system shell to flush all rules:

```
iptables --F
```

If you've already made your `iptables` rules persistent, you will need to repeat the appropriate procedure in the *Make all iptables Rules Persistent* (page 218) section.

14.1.2 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.

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` (page 1072) and `mongos.exe` (page 1073) instances.

The configuration changes outlined in this document will create rules which explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed.

You can configure the *Windows Firewall* with using the `netsh` command line tool or through a windows application. On Windows Server 2008 this application is *Windows Firewall With Advanced Security* in *Administrative Tools*. On previous versions of Windows Server, access the *Windows Firewall* application in the *System and Security* control panel.

The procedures in this document use the `netsh` command line tool.

Patterns

This section contains a number of patterns and examples for configuring *Windows Firewall* for use with MongoDB deployments. If you have configured different ports using the `port` (page 1116) configuration setting, you will need to modify the rules accordingly.

Traffic to and from `mongod.exe` Instances

This pattern is applicable to all `mongod.exe` (page 1072) 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` (page 1072) 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` (page 1072) 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` (page 1073) server, with the following invocation:

```
netsh advfirewall firewall add rule name="Allowing mongos" dir=in action=allow program=" C:\mongodb\b
```

Traffic to and from `mongos.exe` Instances

`mongos.exe` (page 1073) instances provide query routing for *sharded clusters*. Clients connect to `mongos.exe` (page 1073) instances, which behave from the client's perspective as `mongod.exe` (page 1072) instances. In turn, the `mongos.exe` (page 1073) connects to all `mongod.exe` (page 1072) 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` (page 1072) instances that are members of the replica set.

```
netsh advfirewall firewall add rule name="Open mongod shard port 27018" dir=in action=allow protocol=
```

Traffic to and from a MongoDB Config Server

Configuration servers, host the *config database* that stores metadata for sharded clusters. Each production cluster has three configuration servers, initiated using the *mongod --configsvr* option. [4] 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` (page 1073) instances in the cluster *and* all `mongod.exe` (page 1072) 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` (page 1073) instances and the shard `mongod.exe` (page 1072) instances.

[4] You can also run a config server by setting the `configsvr` (page 1125) option in a configuration file.

Traffic to and from a MongoDB Shard Server

For shard servers, running as *mongod --shardsvr* [5] Because the default port number when running with `shardsvr` (page 1125) 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` (page 1072) instances. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members to:

- all `mongod.exe` (page 1072) instances in the shard's replica sets.
- all `mongod.exe` (page 1072) instances in other shards. [6]

Furthermore, shards need to be able make outgoing connections to:

- all `mongos.exe` (page 1073) instances.
- all `mongod.exe` (page 1072) 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` (page 1073) instances:

```
netsh advfirewall firewall add rule name="Open mongod config svr outbound" dir=out action=allow proto
```

Provide Access For Monitoring Systems

1. The `mongostat` (page 1098) diagnostic tool, when running with the `--discover` needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the `mongos.exe` (page 1073) 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` (page 1072) instances running with `shardsvr` (page 1125), 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` (page 1072) instances running with `configsvr` (page 1125), the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongod configsvr HTTP monitoring inbound" dir=in
   ```

[5] You can also specify the shard server option using the `shardsvr` (page 1125) setting in the configuration file. Shard members are also often conventional replica sets using the default port.

[6] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

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`.

Delete all *Windows Firewall* Rules

To delete the firewall rule allowing `mongod.exe` (page 1072) 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
```

List All *Windows Firewall* Rules

To return a list of all *Windows Firewall* rules:

```
netsh advfirewall firewall show rule name=all
```

Reset *Windows Firewall*

To reset the *Windows Firewall* rules:

```
netsh advfirewall reset
```

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"
```

14.1.3 Create a Vulnerability Report

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 the issue so that 10gen can respond appropriately and work to prevent additional issues in the future.

To report an issue, use either jira.mongodb.org (preferred) or email. 10gen responds to vulnerability notifications within 48 hours.

Information to Provide

All vulnerability reports should contain as much information as possible so 10gen 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.

Create the Report in 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/browse/SECURITY/. The ticket number will become the reference identification for the issue for the lifetime of the issue. You can use this identifier for tracking purposes.

Send the Report via Email

While Jira is preferred, you may also report vulnerabilities via email to security@10gen.com.

You may encrypt email using the 10gen public key at http://docs.mongodb.org/10gen-gpg-key.asc.

10gen responds to vulnerability reports sent via email with a response email that contains a reference number for a Jira ticket posted to the SECURITY project.

Evaluation of a Vulnerability Report

10gen validates all submitted vulnerabilities and uses Jira to track all communications regarding a vulnerability, including requests for clarification or additional information. If needed, 10gen representatives set up a conference call to exchange information regarding the vulnerability.

Disclosure

10gen requests that you do *not* publicly disclose any information regarding the vulnerability or exploit the issue until 10gen has had the opportunity to analyze the vulnerability, to respond to the notification, and to notify key users, customers, and partners.

The amount of time required to validate a reported vulnerability depends on the complexity and severity of the issue. 10gen takes all required vulnerabilities very seriously and will always ensure that there is a clear and open channel of communication with the reporter.

After validating an issue, 10gen coordinates public disclosure of the issue with the reporter in a mutually agreed timeframe and format. If required or requested, the reporter of a vulnerability will receive credit in the published security bulletin.

14.2 Access Control

14.2.1 Enable Authentication

Enable authentication using the `auth` (page 1118) or `keyFile` (page 1117) settings. Use `auth` (page 1118) for standalone instances, and `keyFile` (page 1117) with *replica sets* and *sharded clusters*. `keyFile` (page 1117) implies `auth` (page 1118) and allows members of a MongoDB deployment to authenticate internally.

Authentication requires at least one administrator user in the `admin` database. You can create the user before enabling authentication or after enabling authentication.

See also:

Deploy MongoDB with Kerberos Authentication (page 228).

Also consider the *password hashing* (page 240) issue resolved after 2.2.

Procedures

You can enable authentication using either of the following procedures, depending

Create the Administrator Credentials and then Enable Authentication

1. Start the `mongod` (page 1049) or `mongos` (page 1061) instance *without* the `auth` (page 1118) or `keyFile` (page 1117) setting.
2. Create the administrator user as described in *Create a User Administrator* (page 224).
3. Re-start the `mongod` (page 1049) or `mongos` (page 1061) instance with the `auth` (page 1118) or `keyFile` (page 1117) setting.

Enable Authentication and then Create Administrator

1. Start the `mongod` (page 1049) or `mongos` (page 1061) instance with the `auth` (page 1118) or `keyFile` (page 1117) setting.
2. Connect to the instance on the same system so that you can authenticate using the *localhost exception* (page 225).
3. Create the administrator user as described in *Create a User Administrator* (page 224).

Query Authenticated Users

If you have the `userAdmin` (page 235) or `userAdminAnyDatabase` (page 237) role on a database, you can query authenticated users in that database with the following operation:

```
db.system.users.find()
```

14.2.2 Create a User Administrator

In a MongoDB deployment, users with either the `userAdmin` (page 235) or `userAdminAnyDatabase` (page 237) roles are *effective* administrative "superusers". Users with either of these roles can create and modify any other users and can assign them any privileges. The user also can grant *itself* any privileges. In production deployments, this user should have *no other roles* and should only administer users and privileges.

This should be the first user created for a MongoDB deployment. This user can then create all other users in the system.

Important: The `userAdminAnyDatabase` (page 237) user can grant itself and any other user full access to the entire MongoDB instance. The credentials to log in as this user should be carefully controlled.

Users with the `userAdmin` (page 235) and `userAdminAnyDatabase` (page 237) privileges are not the same as the UNIX `root` superuser in that this role confers **no additional access** beyond user administration. These users cannot perform administrative operations or read or write data without first conferring themselves with additional permissions.

Note: The `userAdmin` (page 235) is a database specific privilege, and *only* grants a user the ability to administer users on a single database. However, for the `admin` database, `userAdmin` (page 235) allows a user the ability to gain `userAdminAnyDatabase` (page 237), and so for the `admin` database **only** these roles are effectively the same.

Create a User Administrator

1. Connect to the `mongod` (page 1049) or `mongos` (page 1061) by either:
   - Authenticating as an existing user with the `userAdmin` (page 235) or `userAdminAnyDatabase` (page 237) role.
   - Authenticating using the *localhost exception* (page 225). When creating the first user in a deployment, you must authenticate using the *localhost exception* (page 225).
2. Switch to the `admin` database:

   ```
   db = db.getSiblingDB('admin')
   ```

3. Add the user with either the `userAdmin` (page 235) role or `userAdminAnyDatabase` (page 237) role, and only that role, by issuing a command similar to the following, where `<username>` is the username and `<password>` is the password:

   ```
   db.addUser( { user: "<username>",
                 pwd: "<password>",
                 roles: [ "userAdminAnyDatabase" ] } )
   ```

To authenticate as this user, you must authenticate against the `admin` database.

Authenticate with Full Administrative Access via Localhost

If there are no users for the `admin` database, you can connect with full administrative access via the localhost interface. This bypass exists to support bootstrapping new deployments. This approach is useful, for example, if you want to run `mongod` (page 1049) or `mongos` (page 1061) with authentication before creating your first user.

To authenticate via localhost, connect to the `mongod` (page 1049) or `mongos` (page 1061) from a client running on the same system. Your connection will have full administrative access.

To disable the localhost bypass, set the `enableLocalhostAuthBypass` (page 1129) parameter using `setParameter` (page 1123) during startup:

```
mongod --setParameter enableLocalhostAuthBypass=0
```

Note: For versions of MongoDB 2.2 prior to 2.2.4, if `mongos` (page 1061) is running with `keyFile` (page 1117), then all users connecting over the localhost interface must authenticate, even if there aren't any users in the `admin` database. Connections on localhost are not correctly granted full access on sharded systems that run those versions.

MongoDB 2.2.4 resolves this issue.

Note: In version 2.2, you cannot add the first user to a sharded cluster using the `localhost` connection. If you are running a 2.2 sharded cluster and want to enable authentication, you must deploy the cluster and add the first user to the `admin` database before restarting the cluster to run with `keyFile` (page 1117).

14.2.3 Add a User to a Database

To add a user to a database you must authenticate to that database as a user with the `userAdmin` (page 235) or `userAdminAnyDatabase` (page 237) role. If you have not first created a user with one of those roles, do so as described in *Create a User Administrator* (page 224).

When adding a user to multiple databases, you must define the user *for each database*. See *Password Hashing Insecurity* (page 240) for important security information.

To add a user, pass the `db.addUser()` (page 993) method a well formed *privilege document* (page 233) that contains the user's credentials and privileges. The `db.addUser()` (page 993) method adds the document to the database's `system.users` (page 238) collection.

Changed in version 2.4: In previous versions of MongoDB, you could change an existing user's password by calling `db.addUser()` (page 993) again with the user's username and their updated password. Anything specified in the `addUser()` method would override the existing information for that user. In newer versions of MongoDB, this will result in a duplicate key error.

To change a user's password in version 2.4 or newer, see *Change a User's Password* (page 227).

For the structure of a privilege document, see `system.users` (page 238). For descriptions of user roles, see *User Privilege Roles in MongoDB* (page 233).

Example

The following creates a user named `Alice` in the `products` database and gives her `readWrite` and `dbAdmin` privileges.

```
use products
db.addUser( { user: "Alice",
              pwd: "Moon1234",
              roles: [ "readWrite", "dbAdmin" ]
            } )
```

Example

The following creates a user named `Bob` in the `admin` database. The `privilege document` (page 238) uses Bob's credentials from the `products` database and assigns him `userAdmin` privileges.

```
use admin
db.addUser( { user: "Bob",
              userSource: "products",
              roles: [ "userAdmin" ]
            } )
```

Example

The following creates a user named `Carlos` in the `admin` database and gives him `readWrite` access to the `config` database, which lets him change certain settings for sharded clusters, such as to disable the balancer.

```
db = db.getSiblingDB('admin')
db.addUser( { user: "Carlos",
              pwd: "Moon1234",
              roles: [ "clusterAdmin" ],
              otherDBRoles: { config: [ "readWrite" ]
            } } )
```

Only the `admin` database supports the `otherDBRoles` (page 239) field.

14.2.4 Change a User's Password

New in version 2.4.

To change a user's password, you must have the `userAdmin` (page 235) role on the database that contains the definition of the user whose password you wish to change.

To update the password, pass the user's username and the new desired password to the `db.changeUserPassword()` (page 995) method.

Example

The following operation changes the `reporting` user's password to `SOhSS3TbYhxusooLiW8ypJPxmt1oOfL`:

```
db = db.getSiblingDB('records')
db.changeUserPassword("reporting", "SOhSS3TbYhxusooLiW8ypJPxmt1oOfL")
```

Note: In previous versions of MongoDB, you could change an existing user's password by calling `db.addUser()` (page 993) again with the user's username and their updated password. Anything specified in the `addUser()` method would override the existing information for that user. In newer versions of MongoDB, this will result in a duplicate key error.

For more about changing a user's password prior to version 2.4, see: *Add a User to a Database* (page 226).

14.2.5 Generate a Key File

This section describes how to generate a key file to store authentication information. After generating a key file, specify the key file using the `keyFile` (page 1117) option when starting a `mongod` (page 1049) or `mongos` (page 1061) instance.

A key file must be less than 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. Key file permissions are not checked on Windows systems.

Generate a Key File

Use the following `openssl` command at the system shell to generate pseudo-random content for a key file:

```
openssl rand -base64 741
```

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

Key File Properties

Be aware that MongoDB strips whitespace characters (e.g. `x0d`, `x09`, and `x20`,) for cross-platform convenience. As a result, the following operations produce identical keys:

```
echo -e "my secret key" > key1
echo -e "my secret key\n" > key2
echo -e "my    secret    key" > key3
echo -e "my\r\nsecret\r\nkey\r\n" > key4
```

14.2.6 Deploy MongoDB with Kerberos Authentication

New in version 2.4.

MongoDB Enterprise supports authentication using a Kerberos service to manage the authentication process. Kerberos is an industry standard authentication protocol for large client/server system. With Kerberos MongoDB and application ecosystems can take advantage of existing authentication infrastructure and processes.

Setting up and configuring a Kerberos deployment is beyond the scope of this document. In order to use MongoDB with Kerberos, you must have a properly configured Kerberos deployment and the ability to generate a valid *keytab* file for each `mongod` (page 1049) instance in your MongoDB deployment.

Note: The following assumes that you have a valid Kerberos keytab file for your realm accessible on your system. The examples below assume that the keytab file is valid and is located at `http://docs.mongodb.org/manualopt/mongodb/mongod.keytab` and is *only* accessible to the user that runs the `mongod` (page 1049) process.

Process Overview

To run MongoDB with Kerberos support, you must:

- Configure a Kerberos service principal for each `mongod` (page 1049) and `mongos` (page 1061) instance in your MongoDB deployment.
- Generate and distribute keytab files for each MongoDB component (i.e. `mongod` (page 1049) and `mongos` (page 1061))in your deployment. Ensure that you *only* transmit keytab files over secure channels.
- Optional. Start the `mongod` (page 1049) instance *without* `auth` (page 1118) and create users inside of MongoDB that you can use to bootstrap your deployment.
- Start `mongod` (page 1049) and `mongos` (page 1061) with the `KRB5_KTNAME` environment variable as well as a number of required run time options.
- If you did not create Kerberos user accounts, you can use the *localhost exception* (page 225) to create users at this point until you create the first user on the `admin` database.
- Authenticate clients, including the `mongo` (page 1066) shell using Kerberos.

Operations

Create Users and Privilege Documents

For every user that you want to be able to authenticate using Kerberos, you must create corresponding privilege documents in the `system.users` (page 238) collection to provision access to users. Consider the following document:

```
{
  user: "application/reporting@EXAMPLE.NET",
  roles: ["read"],
  userSource: "$external"
}
```

This grants the Kerberos user principal `application/reporting@EXAMPLE.NET` read only access to a database. The `userSource` (page 239) `$external` reference allows `mongod` (page 1049) to consult an external source (i.e. Kerberos) to authenticate this user.

In the `mongo` (page 1066) shell you can pass the `db.addUser()` (page 993) a user privilege document to provision access to users, as in the following operation:

```
db = db.getSiblingDB("records")
db.addUser( {
              "user": "application/reporting@EXAMPLE.NET",
              "roles": [ "read" ],
              "userSource": "$external"
            } )
```

These operations grants the Kerberos user `application/reporting@EXAMPLE.NET` access to the `records` database.

To remove access to a user, use the `remove()` (page 970) method, as in the following example:

```
db.system.users.remove( { user: "application/reporting@EXAMPLE.NET" } )
```

To modify a user document, use *update* (page 93) operations on documents in the `system.users` (page 238) collection.

See also:

system.users Privilege Documents (page 238) and *User Privilege Roles in MongoDB* (page 233).

Start `mongod` with Kerberos Support

Once you have provisioned privileges to users in the `mongod` (page 1049), *and* obtained a valid keytab file, you must start `mongod` (page 1049) using a command in the following form:

```
env KRB5_KTNAME=<path to keytab file> <mongod invocation>
```

For successful operation with `mongod` (page 1049) use the following run time options in addition to your normal default configuration options:

- *--setParameter* with the `authenticationMechanisms=GSSAPI` argument to enable support for Kerberos.
- *--auth* to enable authentication.
- *--keyFile* to allow components of a single MongoDB deployment to communicate with each other, if needed to support replica set and sharded cluster operations. `keyFile` (page 1117) implies `auth` (page 1118).

For example, consider the following invocation:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
    /opt/mongodb/bin/mongod --dbpath /opt/mongodb/data \
    --fork --logpath /opt/mongodb/log/mongod.log \
    --auth --setParameter authenticationMechanisms=GSSAPI
```

You can also specify these options using the configuration file. As in the following:

```
# /opt/mongodb/mongod.conf, Example configuration file.

fork = true
auth = true

dbpath = /opt/mongodb/data
logpath = /opt/mongodb/log/mongod.log
setParameter = authenticationMechanisms=GSSAPI
```

To use this configuration file, start `mongod` (page 1049) as in the following:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
    /opt/mongodb/bin/mongod --config /opt/mongodb/mongod.conf
```

To start a `mongos` (page 1061) instance using Kerberos, you must create a Kerberos service principal and deploy a keytab file for this instance, and then start the `mongos` (page 1061) with the following invocation:

```
env KRB5_KTNAME=/opt/mongodb/mongos.keytab \
    /opt/mongodb/bin/mongos
    --configdb shard0.example.net,shard1.example.net,shard2.example.net \
    --setParameter authenticationMechanisms=GSSAPI \
    --keyFile /opt/mongodb/mongos.keyfile
```

If you encounter problems when trying to start `mongod` (page 1049) or `mongos` (page 1061), please see the *troubleshooting section* (page 231) for more information.

Important: Before users can authenticate to MongoDB using Kerberos you must *create users* (page 228) and grant them privileges within MongoDB. If you have not created users when you start MongoDB with Kerberos you can use the *localhost authentication exception* (page 225) to add users. See the *Create Users and Privilege Documents* (page 228) section and the *User Privilege Roles in MongoDB* (page 233) document for more information.

Authenticate `mongo` Shell with Kerberos

To connect to a `mongod` (page 1049) instance using the `mongo` (page 1066) shell you must begin by using the `kinit` program to initialize and authenticate a Kerberos session. Then, start a `mongo` (page 1066) instance, and use the `db.auth()` (page 995) method, to authenticate against the special `$external` database, as in the following operation:

```
use $external
db.auth( { mechanism: "GSSAPI", user: "application/reporting@EXAMPLE.NET" } )
```

Alternately, you can authenticate using command line options to `mongo` (page 1066), as in the following equivalent example:

```
mongo --authenticationMechanism=GSSAPI
      --authenticationDatabase='$external' \
      --username application/reporting@EXAMPLE.NET
```

These operations authenticates the Kerberos principal name `application/reporting@EXAMPLE.NET` to the connected `mongod` (page 1049), and will automatically acquire all available privileges as needed.

Use MongoDB Drivers to Authenticate with Kerberos

At the time of release, the C++, Java, C#, and Python drivers all provide support for Kerberos authentication to MongoDB. Consider the following tutorials for more information:

- Java
- C#
- C++
- Python

Troubleshooting

Kerberos Configuration Checklist

If you're having trouble getting `mongod` (page 1049) to start with Kerberos, there are a number of Kerberos-specific issues that can prevent successful authentication. As you begin troubleshooting your Kerberos deployment, ensure that:

- The `mongod` (page 1049) is from MongoDB Enterprise.
- You have a valid keytab file specified in the environment running the `mongod` (page 1049). For the `mongod` (page 1049) instance running on the `db0.example.net` host, the service principal should be `mongodb/db0.example.net`.
- DNS allows the `mongod` (page 1049) to resolve the components of the Kerberos infrastructure. You should have both `A` and `PTR` records (i.e. forward and reverse DNS) for the system that runs the `mongod` (page 1049) instance.
- The canonical system hostname of the system that runs the `mongod` (page 1049) instance is the resolvable fully qualified domain for this host. Test system hostname resolution with the `hostname -f` command at the system prompt.
- Both the Kerberos *KDC* and the system running `mongod` (page 1049) instance must be able to resolve each other using DNS [7]
- The time systems of the systems running the `mongod` (page 1049) instances and the Kerberos infrastructure are synchronized. Time differences greater than 5 minutes will prevent successful authentication.

If you still encounter problems with Kerberos, you can start both `mongod` (page 1049) and `mongo` (page 1066) (or another client) with the environment variable `KRB5_TRACE` set to different files to produce more verbose logging of the Kerberos process to help further troubleshooting, as in the following example:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
    KRB5_TRACE=/opt/mongodb/log/mongodb-kerberos.log \
    /opt/mongodb/bin/mongod --dbpath /opt/mongodb/data \
    --fork --logpath /opt/mongodb/log/mongod.log \
    --auth --setParameter authenticationMechanisms=GSSAPI
```

Common Error Messages

In some situations, MongoDB will return error messages from the GSSAPI interface if there is a problem with the Kerberos service.

`GSSAPI error in client while negotiating security context.`

This error occurs on the client and reflects insufficient credentials or a malicious attempt to authenticate.

If you receive this error ensure that you're using the correct credentials and the correct fully qualified domain name when connecting to the host.

[7] By default, Kerberos attempts to resolve hosts using the content of the `/etc/kerb5.conf` before using DNS to resolve hosts.

`GSSAPI error acquiring credentials.`

This error only occurs when attempting to start the `mongod` (page 1049) or `mongos` (page 1061) and reflects improper configuration of system hostname or a missing or incorrectly configured keytab file. If you encounter this problem, consider all the items in the *Kerberos Configuration Checklist* (page 231), in particular:

- examine the keytab file, with the following command:

  ```
  klist -k <keytab>
  ```

  Replace `<keytab>` with the path to your keytab file.

- check the configured hostname for your system, with the following command:

  ```
  hostname -f
  ```

  Ensure that this name matches the name in the keytab file, or use the `saslHostName` (page 1130) to pass MongoDB the correct hostname.

Enable the Traditional MongoDB Authentication Mechanism

For testing and development purposes you can enable both the Kerberos (i.e. `GSSAPI`) authentication mechanism in combination with the traditional MongoDB challenge/response authentication mechanism (i.e. `MONGODB-CR`), using the following `setParameter` (page 1123) run-time option:

```
mongod --setParameter authenticationMechanisms=GSSAPI,MONGODB-CR
```

> **Warning:** All `keyFile` (page 1117) *internal* authentication between members of a *replica set* or *sharded cluster* still uses the `MONGODB-CR` authentication mechanism, even if `MONGODB-CR` is not enabled. All client authentication will still use Kerberos.

CHAPTER 15

Reference

15.1 User Privilege Roles in MongoDB

New in version 2.4: In version 2.4, MongoDB adds support for the following user roles:

15.1.1 Roles

Changed in version 2.4.

Roles in MongoDB provide users with a set of specific privileges, on specific logical databases. Users may have multiple roles and may have different roles on different logical database. Roles only grant privileges and never limit access: if a user has `read` (page 233) *and* `readWriteAnyDatabase` (page 237) permissions on the `records` database, that user will be able to write data to the `records` database.

Note: By default, MongoDB 2.4 is backwards-compatible with the MongoDB 2.2 access control roles. You can explicitly disable this backwards-compatibility by setting the `supportCompatibilityFormPrivilegeDocuments` (page 1131) option to `0` during startup, as in the following command-line invocation of MongoDB:

```
mongod --setParameter supportCompatibilityFormPrivilegeDocuments=0
```

In general, you should set this option if your deployment does not need to support legacy user documents. Typically legacy user documents are only useful during the upgrade process and while you migrate applications to the updated privilege document form.

See *privilege documents* (page 238) and *Delegated Credentials for MongoDB Authentication* (page 240) for more information about permissions and authentication in MongoDB.

Database User Roles

read

Provides users with the ability to read data from any collection within a specific logical database. This includes `find()` (page 951) and the following *database commands*:

- `aggregate` (page 834)
- `checkShardingIndex` (page 875)
- `cloneCollectionAsCapped` (page 887)
- `collStats` (page 900)

- `count` (page 834)
- `dataSize` (page 906)
- `dbHash` (page 899)
- `dbStats` (page 904)
- `distinct` (page 835)
- `filemd5` (page 888)
- `geoNear` (page 850)
- `geoSearch` (page 851)
- `geoWalk` (page 851)
- `group` (page 836)
- `mapReduce` (page 840) (inline output only.)
- `text` (page 856) (beta feature.)

readWrite

Provides users with the ability to read from or write to any collection within a specific logical database. Users with `readWrite` (page 234) have access to all of the operations available to `read` (page 233) users, as well as the following basic write operations: `insert()` (page 961), `remove()` (page 970), and `update()` (page 974).

Additionally, users with the `readWrite` (page 234) have access to the following *database commands*:

- `cloneCollection` (page 886) (as the target database.)
- `convertToCapped` (page 887)
- `create` (page 885) (and to create collections implicitly.)
- `drop()` (page 948)
- `dropIndexes` (page 888)
- `emptycapped` (page 940)
- `ensureIndex()` (page 949)
- `findAndModify` (page 851)
- `mapReduce` (page 840) (output to a collection.)
- `renameCollection` (page 882) (within the same database.)

Database Administration Roles

dbAdmin

Provides the ability to perform the following set of administrative operations within the scope of this logical database.

- `clean` (page 890)
- `collMod` (page 892)
- `collStats` (page 900)
- `compact` (page 890)
- `convertToCapped` (page 887)

- create (page 885)
- db.createCollection() (page 997)
- dbStats (page 904)
- drop() (page 948)
- dropIndexes (page 888)
- ensureIndex() (page 949)
- indexStats (page 911)
- profile (page 907)
- reIndex (page 894)
- renameCollection (page 882) (within a single database.)
- validate (page 908)

Furthermore, only dbAdmin (page 234) has the ability to read the system.profile (page 1133) collection.

userAdmin

Allows users to read and write data to the system.users (page 238) collection of any database. Users with this role will be able to modify permissions for existing users and create new users. userAdmin (page 235) does not restrict the permissions that a user can grant, and a userAdmin (page 235) user can grant privileges to themselves or other users in excess of the userAdmin (page 235) users' current privileges.

Important: userAdmin (page 235) is effectively the *superuser* role for a specific database. Users with userAdmin (page 235) can grant themselves all privileges. However, userAdmin (page 235) does not explicitly authorize a user for any privileges beyond user administration.

Note: The userAdmin (page 235) is a database specific privilege, and *only* grants a user the ability to administer users on a single database. However, for the admin database, userAdmin (page 235) allows a user the ability to gain userAdminAnyDatabase (page 237), and so for the admin database **only** these roles are effectively the same.

15.1.2 Administrative Roles

clusterAdmin

clusterAdmin (page 235) grants access to several administration operations that affect or present information about the whole system, rather than just a single database. These privileges include but are not limited to *replica set* and *sharded cluster* administrative functions.

clusterAdmin (page 235) is only applicable on the admin database, and does not confer any access to the local or config databases.

Specifically, users with the clusterAdmin (page 235) role have access to the following operations:

- addShard (page 874)
- closeAllDatabases (page 887)
- connPoolStats (page 902)
- connPoolSync (page 890)
- _cpuProfilerStart
- _cpuProfilerStop

- cursorInfo (page 906)
- diagLogging (page 906)
- dropDatabase (page 884)
- enableSharding (page 875)
- flushRouterConfig (page 873)
- fsync (page 888)
- db.fsyncUnlock() (page 1004)
- getCmdLineOpts (page 906)
- getLog (page 916)
- getParameter (page 895)
- getShardMap (page 875)
- getShardVersion (page 876)
- hostInfo (page 917)
- db.currentOp() (page 998)
- db.killOp() (page 1009)
- listDatabases (page 898)
- listShards (page 875)
- logRotate (page 897)
- moveChunk (page 880)
- movePrimary (page 881)
- netstat (page 907)
- removeShard (page 875)
- repairDatabase (page 895)
- replSetFreeze (page 865)
- replSetGetStatus (page 865)
- replSetInitiate (page 866)
- replSetMaintenance (page 867)
- replSetReconfig (page 868)
- replSetStepDown (page 868)
- replSetSyncFrom (page 869)
- resync (page 870)
- serverStatus (page 919)
- setParameter (page 894)
- setShardVersion (page 876)
- shardCollection (page 876)
- shardingState (page 877)

- `shutdown` (page 896)
- `splitChunk` (page 879)
- `splitVector` (page 880)
- `split` (page 878)
- `top` (page 911)
- `touch` (page 896)
- `unsetSharding` (page 877)

For some cluster administration operations, MongoDB requires read and write access to the `local` or `config` databases. You must specify this access separately from `clusterAdmin` (page 235). See the *Combined Access* (page 237) section for more information.

15.1.3 Any Database Roles

Note: You must specify the following "any" database roles on the `admin` databases. These roles apply to all databases in a `mongod` (page 1049) instance and are roughly equivalent to their single-database equivalents.

If you add any of these roles to a *user privilege document* (page 238) outside of the `admin` database, the privilege will have no effect. However, only the specification of the roles must occur in the `admin` database, with *delegated authentication credentials* (page 240), users can gain these privileges by authenticating to another database.

readAnyDatabase
`readAnyDatabase` (page 237) provides users with the same read-only permissions as `read` (page 233), except it applies to *all* logical databases in the MongoDB environment.

readWriteAnyDatabase
`readWriteAnyDatabase` (page 237) provides users with the same read and write permissions as `readWrite` (page 234), except it applies to *all* logical databases in the MongoDB environment.

userAdminAnyDatabase
`userAdminAnyDatabase` (page 237) provides users with the same access to user administration operations as `userAdmin` (page 235), except it applies to *all* logical databases in the MongoDB environment.

Important: Because users with `userAdminAnyDatabase` (page 237) and `userAdmin` (page 235) have the ability to create and modify permissions in addition to their own level of access, this role is *effectively* the MongoDB system superuser. However, `userAdminAnyDatabase` (page 237) and `userAdmin` (page 235) do not explicitly authorize a user for any privileges beyond user administration.

dbAdminAnyDatabase
`dbAdminAnyDatabase` (page 237) provides users with the same access to database administration operations as `dbAdmin` (page 234), except it applies to *all* logical databases in the MongoDB environment.

15.1.4 Combined Access

Some operations are only available to users that have multiple roles. Consider the following:

`sh.status()` (page 1029) Requires `clusterAdmin` (page 235) and `read` (page 233) access to the `config` (page 556) database.

`applyOps` (page 870), `eval` (page 862) [1] Requires `readWriteAnyDatabase` (page 237), `userAdminAnyDatabase` (page 237), `dbAdminAnyDatabase` (page 237) and `clusterAdmin` (page 235) (on the `admin` database.)

Some operations related to cluster administration are not available to users who *only* have the `clusterAdmin` (page 235) role:

`rs.conf()` (page 1015) Requires `read` (page 233) on the `local` database.

`sh.addShard()` (page 1022) Requires `readWrite` (page 234) on the `config` database.

15.2 `system.users` Privilege Documents

Changed in version 2.4.

15.2.1 Overview

The documents in the `<database>.system.users` (page 238) collection store credentials and user privilege information used by the authentication system to provision access to users in the MongoDB system. See *User Privilege Roles in MongoDB* (page 233) for more information about access roles, and *Security* (page 205) for an overview security in MongoDB.

15.2.2 Data Model

`<database>.system.`**`users`**

Changed in version 2.4.

Documents in the `<database>.system.users` (page 238) collection stores credentials and *user roles* (page 233) for users who have access to the database. Consider the following prototypes of user privilege documents:

```
{
   user: "<username>",
   pwd: "<hash>",
   roles: []
}

{
   user: "<username>",
   userSource: "<database>",
   roles: []
}
```

Note: The `pwd` (page 239) and `userSource` (page 239) fields are mutually exclusive. A single document cannot contain both.

The following privilege document with the `otherDBRoles` (page 239) field is only supported on the `admin` database:

```
{
   user: "<username>",
   userSource: "<database>",
   otherDBRoles: {
      <database0> : [],
      <database1> : []
   },
   roles: []
}
```

Consider the content of the following *fields* in the `system.users` (page 238) documents:

`<database>.system.users.`**`user`**

`user` (page 239) is a string that identifies each user. Users exist in the context of a single logical database; however, users from one database may obtain access in another database by way of the `otherDBRoles` (page 239) field on the `admin` database, the `userSource` (page 239) field, or the *Any Database Roles* (page 237).

`<database>.system.users.`**`pwd`**

`pwd` (page 239) holds a *hashed* shared secret used to authenticate the `user` (page 239). `pwd` (page 239) field is mutually exclusive with the `userSource` (page 239) field.

`<database>.system.users.`**`roles`**

`roles` (page 239) holds an array of user roles. The available roles are:

- `read` (page 233)
- `readWrite` (page 234)
- `dbAdmin` (page 234)
- `userAdmin` (page 235)
- `clusterAdmin` (page 235)
- `readAnyDatabase` (page 237)
- `readWriteAnyDatabase` (page 237)
- `userAdminAnyDatabase` (page 237)
- `dbAdminAnyDatabase` (page 237)

See *Roles* (page 233) for full documentation of all available user roles.

`<database>.system.users.`**`userSource`**

A string that holds the name of the database that contains the credentials for the user. If `userSource` (page 239) is `$external`, then MongoDB will use an external resource, such as Kerberos, for authentication credentials.

Note: In the current release, the only external authentication source is Kerberos, which is only available in MongoDB Enterprise.

Use `userSource` (page 239) to ensure that a single user's authentication credentials are only stored in a single location in a `mongod` (page 1049) instance's data.

A `userSource` (page 239) and `user` (page 239) pair identifies a unique user in a MongoDB system.

`admin.system.users.`**`otherDBRoles`**

A document that holds one or more fields with a name that is the name of a database in the MongoDB instance with a value that holds a list of roles this user has on other databases. Consider the following example:

```
{
  user: "admin",
  userSource: "$external",
  roles: [ "clusterAdmin"],
  otherDBRoles:
  {
    config: [ "read" ],
    records: [ "dbadmin" ]
  }
}
```

This user has the following privileges:

- clusterAdmin (page 235) on the admin database,
- read (page 233) on the config (page 556) database, and
- dbAdmin (page 234) on the records database.

15.2.3 Delegated Credentials for MongoDB Authentication

New in version 2.4.

With a new document format in the system.users (page 238) collection, MongoDB now supports the ability to delegate authentication credentials to other sources and databases. The userSource (page 239) field in these documents forces MongoDB to use another source for credentials.

Consider the following document in a system.users (page 238) collection in a database named accounts:

```
{
   user: "application0",
   pwd: "YvuolxMtaycghk2GMrzmImkG4073jzAw2AliMRul",
   roles: []
}
```

Then for *every* database that the application0 user requires access, add documents to the system.users (page 238) collection that resemble the following:

```
{
   user: "application0",
   roles: ['readWrite'],
   userSource: "accounts"
}
```

To gain privileges to databases where the application0 has access, you must first authenticate to the accounts database.

15.2.4 Disable Legacy Privilege Documents

By default MongoDB 2.4 includes support for both new, role-based privilege documents style as well 2.2 and earlier privilege documents. MongoDB assumes any privilege document without a roles (page 239) field is a 2.2 or earlier document.

To ensure that mongod (page 1049) instances will only provide access to users defined with the new role-based privilege documents, use the following setParameter (page 1123) run-time option:

```
mongod --setParameter supportCompatibilityFormPrivilegeDocuments=0
```

15.3 Password Hashing Insecurity

In version 2.2 and earlier:

- the read-write users of a database all have access to the system.users collection, which contains the user names and user password hashes. [2]

  Note: In 2.4, only users with the userAdmin role have access to the system.users collection.

[2] Read-only users do not have access to the system.users collection.

- if a user has the same password for multiple databases, the hash will be the same. A malicious user could exploit this to gain access on a second database using a different user's credentials.

As a result, always use unique username and password combinations for each database.

Thanks to Will Urbanski, from Dell SecureWorks, for identifying this issue.

Part VI

Aggregation

In version 2.2, MongoDB introduced the *aggregation framework* (page 247) that provides a powerful and flexible set of tools to use for many data aggregation tasks. If you're familiar with data aggregation in SQL, consider the *SQL to Aggregation Framework Mapping Chart* (page 287) document as an introduction to some of the basic concepts in the aggregation framework. Consider the full documentation of the aggregation framework and other data aggregation tools for MongoDB here:

CHAPTER 16

Aggregation Framework

New in version 2.1.

16.1 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:

Consider *Aggregation Framework Examples* (page 253) and *Aggregation Framework Reference* (page 263) for more documentation.

16.2 Framework Components

This section provides an introduction to the two concepts that underpin the aggregation framework: *pipelines* and *expressions*.

16.2.1 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 264) to the next to process the documents. Pipeline operators can be repeated in the pipe.

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 263)" includes documentation of the following pipeline operators:

- `$project` (page 264)
- `$match` (page 266)
- `$limit` (page 267)
- `$skip` (page 268)
- `$unwind` (page 268)
- `$group` (page 269)
- `$sort` (page 270)
- `$geoNear` (page 271)

16.2.2 Expressions

Expressions (page 274) 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 269) operator maintain that state (e.g. totals, maximums, minimums, and related data) as documents progress through the *pipeline*.

See also:

Aggregation expressions (page 274) for additional examples of the expressions provided by the aggregation framework.

16.3 Use

16.3.1 Invocation

Invoke an *aggregation* operation with the `aggregate()` (page 945) wrapper in the `mongo` (page 1066) shell or the `aggregate` (page 834) *database command*. Always call `aggregate()` (page 945) on a collection object that determines the input documents of the aggregation *pipeline*. The arguments to the `aggregate()` (page 945) method specify a sequence of *pipeline operators* (page 264), 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 `$project` (page 264) aggregation operator. The `$unwind` (page 268) operator produces one output document per tag. Finally, the `$group` (page 269) operator pivots these fields.

16.3.2 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.

Changed in version 2.4: If an error occurs, the `aggregate()` (page 945) helper throws an exception. In previous versions, the helper returned a document with the error message and code, and `ok` status field not equal to `1`, same as the `aggregate` (page 834) command.

As a document, the result is subject to the *BSON Document size* (page 1139) limit, which is currently 16 megabytes.

16.4 Optimizing Performance

Because you will always call `aggregate` (page 834) 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.

16.4.1 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:

- `$match` (page 266)
- `$sort` (page 270)
- `$limit` (page 267)

- `$skip` (page 268).

The above operators can also use an index when placed **before** the following aggregation operators:

- `$project` (page 264)
- `$unwind` (page 268)
- `$group` (page 269).

New in version 2.4.

The `$geoNear` (page 271) pipeline operator takes advantage of a geospatial index. When using `$geoNear` (page 271), the `$geoNear` (page 271) pipeline operation must appear as the first stage in an aggregation pipeline.

16.4.2 Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the `$match` (page 266) operator to restrict which items go in to the top of the pipeline, as in a query. When placed early in a pipeline, these `$match` (page 266) operations use suitable indexes to scan only the matching documents in a collection.

Placing a `$match` (page 266) pipeline stage followed by a `$sort` (page 270) 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 266) operators at the beginning of the pipeline when possible.

16.4.3 Pipeline Sequence Optimization

Changed in version 2.4.

Aggregation operations have an optimization phase which attempts to re-arrange the pipeline for improved performance.

`$sort` + `$skip` + `$limit` Sequence Optimization

When you have sequence of `$sort` (page 270) followed by a `$skip` (page 268) followed by a `$limit` (page 267), an optimization occurs whereby the `$limit` (page 267) moves in front of the `$skip` (page 268). For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $skip: 10 },
{ $limit: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $limit: 15 }
{ $skip: 10 }
```

Note: The `$limit` (page 267) value has increased to the sum of the initial value and the `$skip` (page 268) value.

`$limit` + `$skip` + `$limit` + `$skip` Sequence Optimization

When you have a continuous sequence of a `$limit` (page 267) pipeline stage followed by a `$skip` (page 268) pipeline stage, the aggregation will attempt to re-arrange the pipeline stages to combine the limits together and the skips together. For example, if the pipeline consists of the following stages:

```
{ $limit: 100 },
{ $skip: 5 },
{ $limit: 10},
{ $skip: 2 }
```

During the intermediate step, the optimizer reverses the position of the `$skip` (page 268) followed by a `$limit` (page 267) to `$limit` (page 267) followed by the `$skip` (page 268).

```
{ $limit: 100 },
{ $limit: 15},
{ $skip: 5 },
{ $skip: 2 }
```

The `$limit` (page 267) value has increased to the sum of the initial value and the `$skip` (page 268) value. Then, for the final `$limit` (page 267) value, the optimizer selects the minimum between the adjacent `$limit` (page 267) values. For the final `$skip` (page 268) value, the optimizer adds the adjacent `$skip` (page 268) values, to transform the sequence to the following:

```
{ $limit: 15 },
{ $skip: 7 }
```

16.4.4 Memory for Cumulative Operators

Certain pipeline operators require access to the entire input set before they can produce any output. For example, `$sort` (page 270) must receive all of the input from the preceding *pipeline* operator before it can produce its first output document. The current implementation of `$sort` (page 270) does not go to disk in these cases: in order to sort the contents of the pipeline, the entire input must fit in memory.

- Changed in version 2.4.
- When a `$sort` (page 270) immediately precedes a `$limit` (page 267) in the pipeline, the `$sort` (page 270) operation only maintains the top n results as it progresses, where n is the specified limit, and MongoDB only needs to store the number of items specified by `$limit` (page 267) in memory. Before MongoDB 2.4, `$sort` (page 270) would sort all the results in memory, and then limit the results to n results.
- Unless the `$sort` (page 270) operator can use an index or immediately precedes a `$limit` (page 267), the `$sort` (page 270) operation must fit within memory. Before MongoDB 2.4, unless the `$sort` (page 270) operator can use an index, the `$sort` (page 270) operation must fit within memory.

`$sort` (page 270) produces an error if the operation consumes 10 percent or more of RAM.

`$group` (page 269) has similar characteristics: Before any `$group` (page 269) passes its output along the pipeline, it must receive the entirety of its input. For the `$group` (page 269) operator, this frequently does not require as much memory as `$sort` (page 270), 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.

16.5 Sharded Operation

Note: Changed in version 2.1.

Some aggregation operations using `aggregate` (page 834) will cause `mongos` (page 1061) 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 `$group` (page 269) or `$sort` (page 270) operation to each shard. [1] Then, a second pipeline on the `mongos` (page 1061) runs. This pipeline consists of the first `$group` (page 269) or `$sort` (page 270) and any remaining pipeline operators, and runs on the results received from the shards.

The `$group` (page 269) operator brings in any "sub-totals" from the shards and combines them: in some cases these may be structures. For example, the `$avg` (page 276) expression maintains a total and count for each shard; `mongos` (page 1061) combines these values and then divides.

16.6 Limitations

Aggregation operations with the `aggregate` (page 834) command have the following limitations:

- The pipeline cannot operate on values of the following types: `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, `CodeWScope`.

  Changed in version 2.4: Removed restriction on `Binary` type data. In 2.2, the pipeline could not operate on `Binary` type data.

- Output from the *pipeline* can only contain 16 megabytes. If your result set exceeds this limit, the `aggregate` (page 834) command produces an error.
- If any single aggregation operation consumes more than 10 percent of system RAM the operation will produce an error.

[1] If an early `$match` (page 266) can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.

CHAPTER 17

Aggregation Framework Examples

MongoDB provides flexible data aggregation functionality with the `aggregate` (page 834) command. For additional information about aggregation consider the following resources:

- *Aggregation Framework* (page 247)
- *Aggregation Framework Reference* (page 263)
- *SQL to Aggregation Framework Mapping Chart* (page 287)

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.

17.1 Requirements

`mongod` (page 1049) and `mongo` (page 1066), version 2.2 or later.

17.2 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` (page 1089) to load this data set into your `mongod` (page 1049) instance.

17.2.1 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()` (page 945) helper in the `mongo` (page 1066) shell. `aggregate()` (page 945) provides a wrapper around the `aggregate` (page 834) database command. See the documentation for your *driver* (page 575) for a more idiomatic interface for data aggregation operations.

17.2.2 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()` (page 945) helper, process all documents on the `zipcodes` collection. `aggregate()` (page 945) connects a number of *pipeline* (page 247) operators, which define the aggregation process.

In the above example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` (page 269) 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 276) operation to calculate the total value of all `pop` fields in the source documents.

  After the `$group` (page 269) operation the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```

- the `$match` (page 266) 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 `$match` (page 266) operation does not alter the documents, which have the same format as the documents output by `$group` (page 269).

The equivalent *SQL* for this operation is:

```
SELECT state, SUM(pop) AS totalPop
       FROM zips
       GROUP BY state
       HAVING totalPop > (10*1000*1000)
```

17.2.3 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()` (page 945) helper, process all documents on the `zipcodes` collection. `aggregate()` (page 945) a number of *pipeline* (page 247) 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 269) 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 `$group` (page 269) operator collects documents by the `state` field and use the `$avg` (page 276) expression to compute a value for the `avgCityPop` field.

The final output of this aggregation operation is:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
},
```

17.2.4 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()` (page 945) helper, process all documents on the `zipcodes` collection. `aggregate()` (page 945) a number of *pipeline* (page 247) 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 269) 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 276) 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 270) 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 269) 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 `$group` (page 269) operator specifies four fields: Using the `$last` (page 274) expression, the `$group` (page 269) operator creates the `biggestcity` and `biggestpop` fields that store the city with the largest population and that population. Using the `$first` (page 274) expression, the `$group` (page 269) 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 264), 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
  }
}
```

17.3 Aggregation with User Preference Data

17.3.1 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"]
}
```

17.3.2 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 `$project` (page 264) operator:
  - creates a new field called `name`.
  - converts the value of the `_id` to upper case, with the `$toUpper` (page 284) operator. Then the `$project` (page 264) creates a new field, named `name` to hold this value.
  - suppresses the `id` field. `$project` (page 264) will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` (page 270) operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

17.3.3 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 `$project` (page 264) operator:
  - Creates two new fields: `month_joined` and `name`.
  - Suppresses the `id` from the results. The `aggregate()` (page 945) method includes the `_id`, unless explicitly suppressed.
- The `$month` (page 285) operator converts the values of the `joined` field to integer representations of the month. Then the `$project` (page 264) operator assigns those values to the `month_joined` field.
- The `$sort` (page 270) 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"
}
```

17.3.4 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 `$project` (page 264) operator creates a new field called `month_joined`.
- The `$month` (page 285) operator converts the values of the `joined` field to integer representations of the month. Then the `$project` (page 264) operator assigns the values to the `month_joined` field.
- The `$group` (page 269) 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, `$group` (page 269) creates a new "per-month" document with two fields:
  - `_id`, which contains a nested document with the `month_joined` field and its value.
  - `number`, which is a generated field. The `$sum` (page 276) operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` (page 270) operator sorts the documents created by `$group` (page 269) 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
}
```

17.3.5 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 268) 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 268) 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 `$group` (page 269) operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` (page 269) creates a new document with two fields:
  - `_id`, which contains the `likes` value.
  - `number`, which is a generated field. The `$sum` (page 276) operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` (page 270) operator sorts these documents by the `number` field in reverse order.
- The `$limit` (page 267) 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 18

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()` (page 945) method. While all examples in this document use this method, `aggregate()` (page 945) is merely a wrapper around the *database command* `aggregate` (page 834). 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()` (page 945) 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 247) overview, the *Aggregation Framework Documentation Index* (page 245), and the *Aggregation Framework Examples* (page 253) for more information on the aggregation functionality.

Aggregation Operators:

18.1 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:

18.1.1 Pipeline Aggregation Operators

Name	Description
`$project` (page 264)	Reshapes a document stream. `$project` (page 264) can rename, add, or remove fields as well as create computed values and sub-documents.
`$match` (page 266)	Filters the document stream, and only allows matching documents to pass into the next pipeline stage. `$match` (page 266) uses standard MongoDB queries.
`$limit` (page 267)	Restricts the number of documents in an aggregation pipeline.
`$skip` (page 268)	Skips over a specified number of documents from the pipeline and returns the rest.
`$unwind` (page 268)	Takes an array of documents and returns them as a stream of documents.
`$group` (page 269)	Groups documents together for the purpose of calculating aggregate values based on a collection of documents.
`$sort` (page 270)	Takes all input documents and returns them in a stream of sorted documents.
`$geoNear` (page 271)	Returns an ordered stream of documents based on proximity to a geospatial point.

$project (aggregation)

`$project`

Reshapes a document stream by renaming, adding, or removing fields. Also use `$project` (page 264) to create computed values or sub-documents. Use `$project` (page 264) to:

- Include fields from the original document.
- Insert computed fields.

- Rename fields.
- Create and populate fields that hold sub-documents.

Use $project (page 264) 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 274). 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 279).

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 264) 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 264) 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 264) 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 279) aggregation expression.

$match (aggregation)

$match

`$match` (page 266) pipes the documents that match its conditions to the next operator in the pipeline.

The `$match` (page 266) query syntax is identical to the *read operation query* (page 42) syntax.

Example

The following operation uses `$match` (page 266) to perform a simple equality match:

```
db.articles.aggregate(
    { $match : { author : "dave" } }
);
```

The `$match` (page 266) selects the documents where the `author` field equals `dave`, and the aggregation returns the following:

```
{ "result" : [
                {
                  "_id" : ObjectId("512bc95fe835e68f199c8686"),
                  "author": "dave",
                  "score" : 80
                },
                { "_id" : ObjectId("512bc962e835e68f199c8687"),
                  "author" : "dave",
                  "score" : 85
                }
              ],
  "ok" : 1 }
```

Example

The following example selects documents to process using the `$match` (page 266) pipeline operator and then pipes the results to the `$group` (page 269) pipeline operator to compute a count of the documents:

```
db.articles.aggregate( [
                        { $match : { score : { $gt : 70, $lte : 90 } } },
                        { $group: { _id: null, count: { $sum: 1 } } }
                       ] );
```

In the aggregation pipeline, `$match` (page 266) selects the documents where the `score` is greater than `70` and less than or equal to `90`. These documents are then piped to the `$group` (page 269) to perform a count. The aggregation returns the following:

```
{
  "result" : [
               {
                 "_id" : null,
                 "count" : 3
               }
             ],
  "ok" : 1 }
```

Note:

- Place the `$match` (page 266) as early in the aggregation *pipeline* as possible. Because `$match` (page 266) limits the total number of documents in the aggregation pipeline, earlier `$match` (page 266) operations minimize the amount of processing down the pipe.
- If you place a `$match` (page 266) at the very beginning of a pipeline, the query can take advantage of *indexes* like any other `db.collection.find()` (page 951) or `db.collection.findOne()` (page 955).

New in version 2.4: `$match` (page 266) queries can support the geospatial `$geoWithin` (page 800) operations.

> **Warning:** You cannot use `$where` (page 797) in `$match` (page 266) queries as part of the aggregation pipeline.

$limit (aggregation)

`$limit`

Restricts the number of *documents* that pass through the `$limit` (page 267) in the *pipeline*.

`$limit` (page 267) 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 267) has no effect on the content of the documents it passes.

Note:

- Changed in version 2.4.
- When a `$sort` (page 270) immediately precedes a `$limit` (page 267) in the pipeline, the `$sort` (page 270) operation only maintains the top n results as it progresses, where n is the specified limit, and MongoDB only needs to store the number of items specified by `$limit` (page 267) in memory. Before

MongoDB 2.4, `$sort` (page 270) would sort all the results in memory, and then limit the results to n results.

•Unless the `$sort` (page 270) operator can use an index or immediately precedes a `$limit` (page 267), the `$sort` (page 270) operation must fit within memory. Before MongoDB 2.4, unless the `$sort` (page 270) operator can use an index, the `$sort` (page 270) operation must fit within memory.

`$sort` (page 270) produces an error if the operation consumes 10 percent or more of RAM.

$skip (aggregation)

$skip

Skips over the specified number of *documents* that pass through the `$skip` (page 268) in the *pipeline* before passing all of the remaining input.

`$skip` (page 268) 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 268) has no effect on the content of the documents it passes along the pipeline.

$unwind (aggregation)

$unwind

Peels off the elements of an array individually, and returns a stream of documents. `$unwind` (page 268) 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 268) operator.

In the above aggregation `$project` (page 264) selects (inclusively) the `author`, `title`, and `tags` fields, as well as the `_id` field implicitly. Then the pipeline passes the results of the projection to the `$unwind` (page 268) 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 268) has the following behaviors:

- `$unwind` (page 268) is most useful in combination with `$group` (page 269).
- You may undo the effects of unwind operation with the `$group` (page 269) pipeline operator.
- If you specify a target field for `$unwind` (page 268) 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 `$unwind` (page 268) that is not an array, `db.collection.aggregate()` (page 945) generates an error.
- If you specify a target field for `$unwind` (page 268) that holds an empty array (`[]`) in an input document, the pipeline ignores the input document, and will generates no result documents.

$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 269) 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 269) cannot output nested documents.

Important: The output of `$group` (page 269) is not ordered.

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 `$project` (page 264) as needed to rename the grouped field after an `$group` (page 269) 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 276) 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 269) must use one of the group aggregation function listed below to generate its composite value:

- `$addToSet` (page 274)
- `$first` (page 274)
- `$last` (page 274)
- `$max` (page 275)
- `$min` (page 275)
- `$avg` (page 276)
- `$push` (page 276)
- `$sum` (page 276)

Warning: The aggregation system currently stores `$group` (page 269) operations in memory, which may cause problems when processing a larger number of groups.

$sort (aggregation)

$sort

The `$sort` (page 270) *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.

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Important: The `$sort` (page 270) cannot begin sorting documents until previous operators in the pipeline have returned all output.

`$sort` (page 270) operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators: `$project` (page 264), `$unwind` (page 268), and `$group` (page 269).

- Changed in version 2.4.
- When a `$sort` (page 270) immediately precedes a `$limit` (page 267) in the pipeline, the `$sort` (page 270) operation only maintains the top n results as it progresses, where n is the specified limit, and MongoDB only needs to store the number of items specified by `$limit` (page 267) in memory. Before MongoDB 2.4, `$sort` (page 270) would sort all the results in memory, and then limit the results to n results.
- Unless the `$sort` (page 270) operator can use an index or immediately precedes a `$limit` (page 267), the `$sort` (page 270) operation must fit within memory. Before MongoDB 2.4, unless the `$sort` (page 270) operator can use an index, the `$sort` (page 270) operation must fit within memory.

`$sort` (page 270) produces an error if the operation consumes 10 percent or more of RAM.

$geoNear (aggregation)

`$geoNear`

New in version 2.4.

`$geoNear` (page 271) returns documents in order of nearest to farthest from a specified point and pass the documents through the aggregation *pipeline*.

Important:

- You can only use `$geoNear` (page 271) as the first stage of a pipeline.
- You must include the `distanceField` option. The `distanceField` option specifies the field that will contain the calculated distance.
- The collection must have a *geospatial index* (page 351).

The `$geoNear` (page 271) accept the following options:

Fields

- **near** (*coordinates*) – Specifies the coordinates (e.g. `[ x, y ]`) to use as the center of a geospatial query.
- **distanceField** (*string*) – Specifies the output field that will contain the calculated distance. You can use the *dot notation* to specify a field within a subdocument.
- **limit** (*number*) – Optional. Specifies the maximum number of documents to return. The default value is 100. See also the `num` option.
- **num** (*number*) – Optional. Synonym for the `limit` option. If both `num` and `limit` are included, the `num` value overrides the `limit` value.
- **maxDistance** (*number*) – Optional. Limits the results to the documents within the specified distance from the center coordinates.
- **query** (*document*) – Optional. Limits the results to the documents that match the query. The query syntax is identical to the *read operation query* (page 42) syntax.
- **spherical** (*boolean*) – Optional. Default value is `false`. When `true`, MongoDB performs calculation using spherical geometry.
- **distanceMultiplier** (*number*) – Optional. Specifies a factor to multiply all distances returned by `$geoNear` (page 271). 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** (*string*) – Optional. Specifies the output field that identifies the location used to calculate the distance. This option is useful when a location field contains multiple locations. You can use the *dot notation* to specify a field within a subdocument.
- **uniqueDocs** (*boolean*) – Optional. Default value is `false`. If a location field contains multiple locations, the default settings will return the document multiple times if more than one location meets the criteria.

  When `true`, the document will only return once even if the document has multiple locations that meet the criteria.

Example

The following aggregation finds at most `5` *unique* documents with a location at most .008 from the center `[40.72, -73.99]` and have `type` equal to `public`:

```
db.places.aggregate([
                      {
                        $geoNear: {
                                    near: [40.724, -73.997],
                                    distanceField: "dist.calculated",
                                    maxDistance: 0.008,
                                    query: { type: "public" },
```

```
                                includeLocs: "dist.location",
                                uniqueDocs: true,
                                num: 5
                              }
                  }
               ])
```

The aggregation returns the following:

```
{
  "result" : [
               { "_id" : 7,
                 "name" : "Washington Square",
                 "type" : "public",
                 "location" : [
                                [ 40.731, -73.999 ],
                                [ 40.732, -73.998 ],
                                [ 40.730, -73.995 ],
                                [ 40.729, -73.996 ]
                              ],
                 "dist" : {
                            "calculated" : 0.0050990195135962296,
                            "location" : [ 40.729, -73.996 ]
                          }
               },
               { "_id" : 8,
                 "name" : "Sara D. Roosevelt Park",
                 "type" : "public",
                 "location" : [
                                [ 40.723, -73.991 ],
                                [ 40.723, -73.990 ],
                                [ 40.715, -73.994 ],
                                [ 40.715, -73.994 ]
                              ],
                 "dist" : {
                            "calculated" : 0.006082762530298062,
                            "location" : [ 40.723, -73.991 ]
                          }
               }
             ],
  "ok" : 1
}
```

The matching documents in the `result` field contain two new fields:

- `dist.calculated` field that contains the calculated distance, and
- `dist.location` field that contains the location used in the calculation.

Note: The options for `$geoNear` (page 271) are similar to the `geoNear` (page 850) command with the following exceptions:

- `distanceField` is a mandatory field for the `$geoNear` (page 271) pipeline operator; the option does not exist in the `geoNear` (page 850) command.
- `includeLocs` accepts a `string` in the `$geoNear` (page 271) pipeline operator and a `boolean` in the `geoNear` (page 850) command.

18.2 Expressions

These operators calculate values within the *aggregation framework*.

18.2.1 `$group` Operators

The `$group` (page 269) pipeline stage provides the following operations:

Group Aggregation Operators

Name	Description
`$addToSet` (page 274)	Returns an array of all the *unique* values for the selected field among for each document in that group.
`$first` (page 274)	Returns the first value in a group.
`$last` (page 274)	Returns the last value in a group.
`$max` (page 275)	Returns the highest value in a group.
`$min` (page 275)	Returns the lowest value in a group.
`$avg` (page 276)	Returns an average of all the values in a group.
`$push` (page 276)	Returns an array of *all* values for the selected field among for each document in that group.
`$sum` (page 276)	Returns the sum of all the values in a group.

$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.

$first (aggregation)

`$first`

Returns the first value it encounters for its group.

Note: Only use `$first` (page 274) when the `$group` (page 269) follows an `$sort` (page 270) operation. Otherwise, the result of this operation is unpredictable.

$last (aggregation)

`$last`

Returns the last value it encounters for its group.

Note: Only use `$last` (page 274) when the `$group` (page 269) follows an `$sort` (page 270) operation. Otherwise, the result of this operation is unpredictable.

$max (aggregation)

$max

Returns the highest value among all values of the field in all documents selected by this group.

$min (aggregation)

$min

The $min (page 275) operator returns the lowest non-null value of a field in the documents for a $group (page 269) operation.

Changed in version 2.4: If some, **but not all**, documents for the $min (page 275) operation have either a null value for the field or are missing the field, the $min (page 275) operator only considers the non-null and the non-missing values for the field. If **all** documents for the $min (page 275) operation have null value for the field or are missing the field, the $min (page 275) operator returns null for the minimum value.

Before 2.4, if any of the documents for the $min (page 275) operation were missing the field, the $min (page 275) operator would not return any value. If any of the documents for the $min (page 275) had the value null, the $min (page 275) operator would return a null.

Example

The users collection contains the following documents:

```
{ "_id" : "abc001", "age" : 25 }
{ "_id" : "abe001", "age" : 35 }
{ "_id" : "efg001", "age" : 20 }
{ "_id" : "xyz001", "age" : 15 }
```

- To find the minimum value of the age field from all the documents, use the $min (page 275) operator:

  ```
  db.users.aggregate( [ { $group: { _id:0, minAge: { $min: "$age"} } } ] )
  ```

  The operation returns the value of the age field in the minAge field:

  ```
  { "result" : [ { "_id" : 0, "minAge" : 15 } ], "ok" : 1 }
  ```

- To find the minimum value of the age field for only those documents with _id starting with the letter a, use the $min (page 275) operator after a $match (page 266) operation:

  ```
  db.users.aggregate( [ { $match: { _id: /^a/ } },
                        { $group: { _id: 0, minAge: { $min: "$age"} } }
                      ] )
  ```

  The operation returns the minimum value of the age field for the two documents with _id starting with the letter a:

  ```
  { "result" : [ { "_id" : 0, "minAge" : 25 } ], "ok" : 1 }
  ```

Example

The users collection contains the following documents where some of the documents are either missing the age field or the age field contains null:

```
{ "_id" : "abc001", "age" : 25 }
{ "_id" : "abe001", "age" : 35 }
{ "_id" : "efg001", "age" : 20 }
```

```
{ "_id" : "xyz001", "age" : 15 }
{ "_id" : "xxx001" }
{ "_id" : "zzz001", "age" : null }
```

- The following operation finds the minimum value of the `age` field in all the documents:

  ```
  db.users.aggregate( [ { $group: { _id:0, minAge: { $min: "$age"} } } ] )
  ```

  Because only some documents for the `$min` (page 275) operation are missing the `age` field or have `age` field equal to `null`, `$min` (page 275) only considers the non-null and the non-missing values and the operation returns the following document:

  ```
  { "result" : [ { "_id" : 0, "minAge" : 15 } ], "ok" : 1 }
  ```

- The following operation finds the minimum value of the `age` field for only those documents where the `_id` equals `"xxx001"` or `"zzz001"`:

  ```
  db.users.aggregate( [ { $match: { _id: {$in: [ "xxx001", "zzz001" ] } } },
                        { $group: { _id: 0, minAge: { $min: "$age"} } }
                      ] )
  ```

  The `$min` (page 275) operation returns `null` for the minimum age since **all** documents for the `$min` (page 275) operation have `null` value for the field `age` or are missing the field:

  ```
  { "result" : [ { "_id" : 0, "minAge" : null } ], "ok" : 1 }
  ```

$avg (aggregation)

$avg

Returns the average of all the values of the field in all documents selected by this group.

$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.

$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 276) 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.

18.2.2 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.

Boolean Aggregation Operators

Name	Description
$and (page 277)	Returns true only when *all* values in its input array are true.
$or (page 277)	Returns true when *any* value in its input array are true.
$not (page 277)	Returns the boolean value that is the opposite of the input value.

$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 (page 277) returns false.

Note: $and (page 277) uses short-circuit logic: the operation stops evaluation after encountering the first false expression.

$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 (page 277) returns false.

Note: $or (page 277) uses short-circuit logic: the operation stops evaluation after encountering the first true expression.

$not (aggregation)

$not

Returns the boolean opposite value passed to it. When passed a true value, $not (page 277) returns false; when passed a false value, $not (page 277) returns true.

18.2.3 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 278), all comparison operators return a Boolean value. $cmp (page 278) returns an integer.

Comparison Aggregation Operators

Name	Description
$cmp (page 278)	Compares two values and returns the result of the comparison as an integer.
$eq (page 278)	Takes two values and returns true if the values are equivalent.
$gt (page 278)	Takes two values and returns true if the first is larger than the second.
$gte (page 278)	Takes two values and returns true if the first is larger than or equal to the second.
$lt (page 278)	Takes two values and returns true if the second value is larger than the first.
$lte (page 279)	Takes two values and returns true if the second value is larger than or equal to the first.
$ne (page 279)	Takes two values and returns true if the values are *not* equivalent.

$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.

$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.

$gt (aggregation)

$gt

Takes two values in an array and returns a boolean. 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 a boolean. 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 (aggregation)

$lt

Takes two values in an array and returns a boolean. 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 a boolean. 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 (aggregation)

$ne

Takes two values in an array returns a boolean. The returned value is:

- `true` when the values are **not equivalent**.
- `false` when the values are **equivalent**.

18.2.4 Arithmetic Operators

These operators only support numbers.

Arithmetic Aggregation Operators

Name	Description
`$add` (page 279)	Computes the sum of an array of numbers.
`$divide` (page 279)	Takes two numbers and divides the first number by the second.
`$mod` (page 280)	Takes two numbers and calcualtes the modulo of the first number divided by the second.
`$multiply` (page 280)	Computes the product of an array of numbers.
`$subtract` (page 280)	Takes two numbers and subtracts the second number from the first.

$add (aggregation)

$add

Takes an array of one or more numbers and adds them together, returning the sum.

$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.

$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 (page 794)

$multiply (aggregation)

$multiply

Takes an array of one or more numbers and multiples them, returning the resulting product.

$subtract (aggregation)

$subtract

Takes an array that contains a pair of numbers and subtracts the second from the first, returning their difference.

18.2.5 String Operators

These operators manipulate strings within projection expressions.

String Aggregation Operators

Name	Description
$concat (page 280)	Concatenates two strings.
$strcasecmp (page 283)	Compares two strings and returns an integer that reflects the comparison.
$substr (page 283)	Takes a string and returns portion of that string.
$toLower (page 283)	Converts a string to lowercase.
$toUpper (page 284)	Converts a string to uppercase.

$concat (aggregation)

$concat

New in version 2.4.

Takes an array of strings, concatenates the strings, and returns the concatenated string. $concat (page 280) can only accept an array of strings.

Use $concat (page 280) with the following syntax:

```
{ $concat: [ <string>, <string>, ... ] }
```

If array element has a value of `null` or refers to a field that is missing, $concat (page 280) will return `null`.

Example

Project new concatenated values.

A collection `menu` contains the documents that stores information on menu items separately in the `section`, the `category` and the `type` fields, as in the following:

```
{ _id: 1, item: { sec: "dessert", category: "pie", type: "apple" } }
{ _id: 2, item: { sec: "dessert", category: "pie", type: "cherry" } }
{ _id: 3, item: { sec: "main", category: "pie", type: "shepherd's" } }
{ _id: 4, item: { sec: "main", category: "pie", type: "chicken pot" } }
```

The following operation uses `$concat` (page 280) to concatenate the `type` field from the sub-document `item`, a space, and the `category` field from the sub-document `item` to project a new `food` field:

```
db.menu.aggregate( { $project: { food:
                                   { $concat: [ "$item.type",
                                                " ",
                                                "$item.category"
                                              ]
                                   }
                               }
                   }
                 )
```

The operation returns the following result set where the `food` field contains the concatenated strings:

```
{
  "result" : [
               { "_id" : 1, "food" : "apple pie" },
               { "_id" : 2, "food" : "cherry pie" },
               { "_id" : 3, "food" : "shepherd's pie" },
               { "_id" : 4, "food" : "chicken pot pie" }
             ],
  "ok" : 1
}
```

Example

Group by a concatenated string.

A collection `menu` contains the documents that stores information on menu items separately in the `section`, the `category` and the `type` fields, as in the following:

```
{ _id: 1, item: { sec: "dessert", category: "pie", type: "apple" } }
{ _id: 2, item: { sec: "dessert", category: "pie", type: "cherry" } }
{ _id: 3, item: { sec: "main", category: "pie", type: "shepherd's" } }
{ _id: 4, item: { sec: "main", category: "pie", type: "chicken pot" } }
```

The following aggregation uses `$concat` (page 280) to concatenate the `sec` field from the sub-document `item`, the string `": "`, and the `category` field from the sub-document `item` to group by the new concatenated string and perform a count:

```
db.menu.aggregate( { $group: { _id:
                                 { $concat: [ "$item.sec",
                                              ": ",
                                              "$item.category"
                                            ]
                                 },
                               count: { $sum: 1 }
                             }
                   }
                 )
```

The aggregation returns the following document:

```
{
  "result" : [
               { "_id" : "main: pie", "count" : 2 },
               { "_id" : "dessert: pie", "count" : 2 }
             ],
  "ok" : 1
}
```

Example

Concatenate `null` or missing values.

A collection `menu` contains the documents that stores information on menu items separately in the `section`, the `category` and the `type` fields. Not all documents have the all three fields. For example, the document with `_id` equal to `5` is missing the `category` field:

```
{ _id: 1, item: { sec: "dessert", category: "pie", type: "apple" } }
{ _id: 2, item: { sec: "dessert", category: "pie", type: "cherry" } }
{ _id: 3, item: { sec: "main", category: "pie", type: "shepherd's" } }
{ _id: 4, item: { sec: "main", category: "pie", type: "chicken pot" } }
{ _id: 5, item: { sec: "beverage", type: "coffee" } }
```

The following aggregation uses the `$concat` (page 280) to concatenate the `type` field from the sub-document `item`, a space, and the `category` field from the sub-document `item`:

```
db.menu.aggregate( { $project: { food:
                                       { $concat: [ "$item.type",
                                                    " ",
                                                    "$item.category"
                                                  ]
                                       }
                               }
                   }
                 )
```

Because the document with `_id` equal to `5` is missing the `type` field in the `item` sub-document, `$concat` (page 280) returns the value `null` as the concatenated value for the document:

```
{
  "result" : [
               { "_id" : 1, "food" : "apple pie" },
               { "_id" : 2, "food" : "cherry pie" },
               { "_id" : 3, "food" : "shepherd's pie" },
               { "_id" : 4, "food" : "chicken pot pie" },
               { "_id" : 5, "food" : null }
             ],
  "ok" : 1
}
```

To handle possible missing fields, you can use `$ifNull` (page 286) with `$concat` (page 280), as in the following example which substitutes `<unknown type>` if the field `type` is `null` or missing, and `<unknown category>` if the field `category` is `null` or is missing:

```
db.menu.aggregate( { $project: { food:
                                       { $concat: [ { $ifNull: ["$item.type", "<unknown type>"]
                                                    " ",
                                                    { $ifNull: ["$item.category", "<unknown cate
                                                  ]
```

```
                                        }
                                }
                        }
                )
```

The aggregation returns the following result set:

```
{
  "result" : [
                { "_id" : 1, "food" : "apple pie" },
                { "_id" : 2, "food" : "cherry pie" },
                { "_id" : 3, "food" : "shepherd's pie" },
                { "_id" : 4, "food" : "chicken pot pie" },
                { "_id" : 5, "food" : "coffee <unknown category>" }
              ],
  "ok" : 1
}
```

$strcasecmp (aggregation)

$strcasecmp

Takes in two strings. Returns a number. `$strcasecmp` (page 283) is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. `$strcasecmp` (page 283) returns 0 if the strings are identical.

Note: `$strcasecmp` (page 283) may not make sense when applied to glyphs outside the Roman alphabet.

`$strcasecmp` (page 283) internally capitalizes strings before comparing them to provide a case-*insensitive* comparison. Use `$cmp` (page 278) for a case sensitive comparison.

$substr (aggregation)

$substr

`$substr` (page 283) 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 283) is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$toLower (aggregation)

$toLower

Takes a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase.

Note: `$toLower` (page 283) 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 284) may not make sense when applied to glyphs outside the Roman alphabet.

18.2.6 Date Operators

All date operators take a "Date" typed value as a single argument and return a number.

Date Aggregation Operators

Name	Description
`$dayOfYear` (page 284)	Converts a date to a number between 1 and 366.
`$dayOfMonth` (page 284)	Converts a date to a number between 1 and 31.
`$dayOfWeek` (page 284)	Converts a date to a number between 1 and 7.
`$year` (page 285)	Converts a date to the full year.
`$month` (page 285)	Converts a date into a number between 1 and 12.
`$week` (page 285)	Converts a date into a number between 0 and 53
`$hour` (page 285)	Converts a date into a number between 0 and 23.
`$minute` (page 285)	Converts a date into a number between 0 and 59.
`$second` (page 285)	Converts a date into a number between 0 and 59. May be 60 to account for leap seconds.
`$millisecond` (page 285)	Returns the millisecond portion of a date as an integer between 0 and 999.

$dayOfYear (aggregation)

$dayOfYear

Takes a date and returns the day of the year as a number between 1 and 366.

$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.)

$year (aggregation)

$year
Takes a date and returns the full year.

$month (aggregation)

$month
Takes a date and returns the month as a number between 1 and 12.

$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.

$hour (aggregation)

$hour
Takes a date and returns the hour between 0 and 23.

$minute (aggregation)

$minute
Takes a date and returns the minute between 0 and 59.

$second (aggregation)

$second
Takes a date and returns the second between 0 and 59, but can be 60 to account for leap seconds.

$millisecond (aggregation)

$millisecond
Takes a date and returns the millisecond portion of the date as an integer between 0 and 999.

18.2.7 Conditional Expressions

Conditional Aggregation Operators

Name	Description
`$cond` (page 286)	Evaluates a an expression and returns a boolean.
`$ifNull` (page 286)	Evaluates an expression and returns a value.

$cond (aggregation)

$cond

Use the $cond (page 286) 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 286) returns the value of the second expression. If the first expression evaluates to false, $cond (page 286) evaluates and returns the third expression.

$ifNull (aggregation)

$ifNull

Use the $ifNull (page 286) operator with the following syntax:

```
{ $ifNull: [ <expression>, <replacement-if-null> ] }
```

Takes an array with two expressions. $ifNull (page 286) returns the first expression if it evaluates to a non-null value. Otherwise, $ifNull (page 286) returns the second expression's value.

SQL to Aggregation Framework Mapping Chart

The *aggregation framework* (page 247) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL. If you're new to MongoDB you might want to consider the *Frequently Asked Questions* (page 731) section for a selection of common questions.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators* (page 264):

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match` (page 266)
GROUP BY	`$group` (page 269)
HAVING	`$match` (page 266)
SELECT	`$project` (page 264)
ORDER BY	`$sort` (page 270)
LIMIT	`$limit` (page 267)
SUM()	`$sum`
COUNT()	`$sum`
join	No direct corresponding operator; *however*, the `$unwind` (page 268) operator allows for somewhat similar functionality, but with fields embedded within the document.

19.1 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.

SELECT cust_id,	db.orders.aggregate([	For each unique cust_id, sum the

Map-Reduce

Map-reduce operations can handle complex aggregation tasks. To perform map-reduce operations, MongoDB provides the `mapReduce` (page 840) command and, in the `mongo` (page 1066) shell, the `db.collection.mapReduce()` (page 963) wrapper method.

20.1 Examples

For examples of map-reduce, see

20.1.1 Map-Reduce Examples

In the `mongo` (page 1066) shell, the `db.collection.mapReduce()` (page 963) method is a wrapper around the `mapReduce` (page 840) command. The following examples use the `db.collection.mapReduce()` (page 963) method:

Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer

Perform map-reduce operation on the `orders` collection to group by the `cust_id`, and for each `cust_id`, calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:

   - In the function, `this` refers to the document that the map-reduce operation is processing.

- The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

```
var mapFunction1 = function() {
                       emit(this.cust_id, this.price);
                   };
```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:

- The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
- The function reduces the `valuesPrice` array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                          return Array.sum(valuesPrices);
                      };
```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

```
db.orders.mapReduce(
                     mapFunction1,
                     reduceFunction1,
                     { out: "map_reduce_example" }
                   )
```

This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item

In this example you will perform a map-reduce operation on the `orders` collection, for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and for each `sku` calculates the number of orders and the total quantity ordered. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

- In the function, `this` refers to the document that the map-reduce operation is processing.
- For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
                                         qty: this.items[idx].qty
                                       };
                           emit(key, value);
                       }
                    };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

- `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.

- The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
- In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
                     reducedVal = { count: 0, qty: 0 };

                     for (var idx = 0; idx < countObjVals.length; idx++) {
                         reducedVal.count += countObjVals[idx].count;
                         reducedVal.qty += countObjVals[idx].qty;
                     }

                     return reducedVal;
                  };
```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

                       reducedVal.avg = reducedVal.qty/reducedVal.count;

                       return reducedVal;

                    };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                       out: { merge: "map_reduce_example" },
                       query: { ord_date:
                                  { $gt: new Date('01/01/2012') }
                              },
                       finalize: finalizeFunction2
                     }
                   )
```

This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

20.1.2 Perform Incremental Map-Reduce

Map-reduce operations can handle complex aggregation tasks. To perform map-reduce operations, MongoDB provides the `mapReduce` (page 840) command and, in the `mongo` (page 1066) shell, the `db.collection.mapReduce()` (page 963) wrapper method.

If the map-reduce data set is constantly growing, then rather than performing the map-reduce operation over the entire data set each time you want to run map-reduce, you may want to perform an incremental map-reduce.

To perform incremental map-reduce:

1. Run a map-reduce job over the current collection and output the result to a separate collection.

2. When you have more data to process, run subsequent map-reduce job with:
   - the `query` parameter that specifies conditions that match *only* the new documents.
   - the `out` parameter that specifies the `reduce` action to merge the new results into the existing output collection.

Consider the following example where you schedule a map-reduce operation on a `sessions` collection to run at the end of each day.

Data Setup

The `sessions` collection contains documents that log users' session each day, for example:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 } );

db.sessions.save( { userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 } );
```

Initial Map-Reduce of Current Collection

Run the first map-reduce operation as follows:

1. Define the `map` function that maps the `userid` to an object that contains the fields `userid`, `total_time`, `count`, and `avg_time`:

```
var mapFunction = function() {
                      var key = this.userid;
                      var value = {
                                    userid: this.userid,
                                    total_time: this.length,
                                    count: 1,
                                    avg_time: 0
                                   };

                      emit( key, value );
                  };
```

2. Define the corresponding `reduce` function with two arguments `key` and `values` to calculate the total time and the count. The `key` corresponds to the `userid`, and the `values` is an array whose elements corresponds to the individual objects mapped to the `userid` in the `mapFunction`.

```
var reduceFunction = function(key, values) {

                        var reducedObject = {
                                              userid: key,
                                              total_time: 0,
                                              count:0,
                                              avg_time:0
                                            };

                        values.forEach( function(value) {
                                              reducedObject.total_time += value.total_time;
```

```
                                          reducedObject.count += value.count;
                                    }
                                  );
                         return reducedObject;
                      };
```

3. Define `finalize` function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` document to add another field `average` and returns the modified document.

```
var finalizeFunction = function (key, reducedValue) {

                          if (reducedValue.count > 0)
                              reducedValue.avg_time = reducedValue.total_time / reducedValue.cou

                          return reducedValue;
                       };
```

4. Perform map-reduce on the `session` collection using the `mapFunction`, the `reduceFunction`, and the `finalizeFunction` functions. Output the results to a collection `session_stat`. If the `session_stat` collection already exists, the operation will replace the contents:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         out: { reduce: "session_stat" },
                         finalize: finalizeFunction
                       }
                     )
```

Subsequent Incremental Map-Reduce

Later as the `sessions` collection grows, you can run additional map-reduce operations. For example, add new documents to the `sessions` collection:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-05 14:17:00'), length: 100 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-05 14:23:00'), length: 115 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-05 15:02:00'), length: 125 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-05 16:45:00'), length: 55 } );
```

At the end of the day, perform incremental map-reduce on the `sessions` collection but use the `query` field to select only the new documents. Output the results to the collection `session_stat`, but `reduce` the contents with the results of the incremental map-reduce:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         query: { ts: { $gt: ISODate('2011-11-05 00:00:00') } },
                         out: { reduce: "session_stat" },
                         finalize: finalizeFunction
                       }
                     );
```

For many simple aggregation tasks, see the *aggregation framework* (page 247).

20.2 Temporary Collection

The map-reduce operation uses a temporary collection during processing. At completion, the map-reduce operation renames the temporary collection. As a result, you can perform a map-reduce operation periodically with the same target collection name without affecting the intermediate states. Use this mode when generating statistical output collections on a regular basis.

20.3 Concurrency

The map-reduce operation is composed of many tasks, including:

- reads from the input collection,
- executions of the `map` function,
- executions of the `reduce` function,
- writes to the output collection.

These various tasks take the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The insert into the temporary collection takes a write lock for a single write.

  If the output collection does not exist, the creation of the output collection takes a write lock.

  If the output collection exists, then the output actions (i.e. `merge`, `replace`, `reduce`) take a write lock.

Changed in version 2.4: The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, JavaScript code (i.e. `map`, `reduce`, `finalize` functions) executed in a single thread.

Note: The final write lock during post-processing makes the results appear atomically. However, output actions `merge` and `reduce` may take minutes to process. For the `merge` and `reduce`, the `nonAtomic` flag is available. See the `db.collection.mapReduce()` (page 963) reference for more information.

20.4 Sharded Cluster

20.4.1 Sharded Input

When using sharded collection as the input for a map-reduce operation, `mongos` (page 1061) will automatically dispatch the map-reduce job to each shard in parallel. There is no special option required. `mongos` (page 1061) will wait for jobs on all shards to finish.

20.4.2 Sharded Output

By default the output collection is not sharded. The process is:

- `mongos` (page 1061) dispatches a map-reduce finish job to the shard that will store the target collection.
- The target shard pulls results from all other shards, and runs a final reduce/finalize operation, and write to the output.

- If using the `sharded` option to the `out` parameter, MongoDB shards the output using `_id` field as the shard key.

  Changed in version 2.2.

- If the output collection does not exist, MongoDB creates and shards the collection on the `_id` field. If the collection is empty, MongoDB creates *chunks* using the result of the first stage of the map-reduce operation.
- `mongos` (page 1061) dispatches, in parallel, a map-reduce finish job to every shard that owns a chunk.
- Each shard will pull the results it owns from all other shards, run a final reduce/finalize, and write to the output collection.

Note:

- During later map-reduce jobs, MongoDB splits chunks as needed.
- Balancing of chunks for the output collection is automatically prevented during post-processing to avoid concurrency issues.

In MongoDB 2.0:

- `mongos` (page 1061) retrieves the results from each shard, and performs merge sort to order the results, and performs a reduce/finalize as needed. `mongos` (page 1061) then writes the result to the output collection in sharded mode.
- This model requires only a small amount of memory, even for large data sets.
- Shard chunks are not automatically split during insertion. This requires manual intervention until the chunks are granular and balanced.

Warning: For best results, only use the sharded output options for `mapReduce` (page 840) in version 2.2 or later.

20.5 Troubleshooting Map-Reduce Operations

You can troubleshoot the `map` function and the `reduce` function in the `mongo` (page 1066) shell. See the following tutorials for more information:

20.5.1 Troubleshoot the Map Function

The `map` function is a JavaScript function that associates or "maps" a value with a key and emits the key and value pair during a *map-reduce* (page 291) operation.

To verify the `key` and `value` pairs emitted by the `map` function, write your own `emit` function.

Consider a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

1. Define the map function that maps the price to the cust_id for each document and emits the cust_id and price pair:

```
var map = function() {
    emit(this.cust_id, this.price);
};
```

2. Define the emit function to print the key and value:

```
var emit = function(key, value) {
    print("emit");
    print("key: " + key + "  value: " + tojson(value));
}
```

3. Invoke the map function with a single document from the orders collection:

```
var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
map.apply(myDoc);
```

4. Verify the key and value pair is as you expected.

```
emit
key: abc123 value:250
```

5. Invoke the map function with multiple documents from the orders collection:

```
var myCursor = db.orders.find( { cust_id: "abc123" } );

while (myCursor.hasNext()) {
    var doc = myCursor.next();
    print ("document _id= " + tojson(doc._id));
    map.apply(doc);
    print();
}
```

6. Verify the key and value pairs are as you expected.

See also:

The map function must meet various requirements. For a list of all the requirements for the map function, see mapReduce (page 840), or the mongo (page 1066) shell helper method db.collection.mapReduce() (page 963).

20.5.2 Troubleshoot the Reduce Function

The reduce function is a JavaScript function that "reduces" to a single object all the values associated with a particular key during a *map-reduce* (page 291) operation. The reduce function must meet various requirements. This tutorial helps verify that the reduce function meets the following criteria:

- The reduce function must return an object whose *type* must be **identical** to the type of the value emitted by the map function.
- The order of the elements in the valuesArray should not affect the output of the reduce function.
- The reduce function must be *idempotent*.

For a list of all the requirements for the reduce function, see mapReduce (page 840), or the mongo (page 1066) shell helper method db.collection.mapReduce() (page 963).

Confirm Output Type

You can test that the `reduce` function returns a value that is the same type as the value emitted from the `map` function.

1. Define a `reduceFunction1` function that takes the arguments `keyCustId` and `valuesPrices`. `valuesPrices` is an array of integers:

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                          return Array.sum(valuesPrices);
                      };
```

2. Define a sample array of integers:

```
var myTestValues = [ 5, 5, 10 ];
```

3. Invoke the `reduceFunction1` with `myTestValues`:

```
reduceFunction1('myKey', myTestValues);
```

4. Verify the `reduceFunction1` returned an integer:

```
20
```

5. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

6. Define a sample array of documents:

```
var myTestObjects = [
                      { count: 1, qty: 5 },
                      { count: 2, qty: 10 },
                      { count: 3, qty: 15 }
                    ];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Insensitivity to the Order of Mapped Values

The `reduce` function takes a `key` and a `values` array as its argument. You can test that the result of the `reduce` function does not depend on the order of the elements in the `values` array.

1. Define a sample `values1` array and a sample `values2` array that only differ in the order of the array elements:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];

var values2 = [
                { count: 3, qty: 15 },
                { count: 1, qty: 5 },
                { count: 2, qty: 10 }
              ];
```

2. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

3. Invoke the `reduceFunction2` first with `values1` and then with `values2`:

```
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Reduce Function Idempotence

Because the map-reduce operation may call a `reduce` multiple times for the same key, and won't call a `reduce` for single instances of a key in the working set, the `reduce` function must return a value of the same type as the value emitted from the `map` function. You can test that the `reduce` function process "reduced" values without affecting the *final* value.

1. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

2. Define a sample key:

```
var myKey = 'myKey';
```

3. Define a sample `valuesIdempotent` array that contains an element that is a call to the `reduceFunction2` function:

```
var valuesIdempotent = [
                         { count: 1, qty: 5 },
                         { count: 2, qty: 10 },
                         reduceFunction2(myKey, [ { count:3, qty: 15 } ] )
                       ];
```

4. Define a sample `values1` array that combines the values passed to `reduceFunction2`:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];
```

5. Invoke the `reduceFunction2` first with `myKey` and `valuesIdempotent` and then with `myKey` and `values1`:

```
reduceFunction2(myKey, valuesIdempotent);
reduceFunction2(myKey, values1);
```

6. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

CHAPTER 21

Simple Aggregation Methods and Commands

In addition to the *aggregation framework* (page 247) and *map-reduce*, MongoDB provides the following methods and commands to perform aggregation:

21.1 Count

MongoDB offers the following command and methods to provide `count` functionality:

- *count* (page 834)
- *db.collection.count()* (page 946)
- *cursor.count()* (page 978)

21.2 Distinct

MongoDB offers the following command and method to provide the `distinct` functionality:

- *distinct* (page 835)
- *db.collection.distinct()* (page 948)

21.3 Group

MongoDB offers the following command and method to provide `group` functionality:

- *group* (page 836)
- *db.collection.group()* (page 958)

Also consider the *Aggregation Commands* (page 833) table which provides reference for and a comparison of the main aggregation operations in MongoDB.

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 309). For procedures and operational concerns, see *Indexing Operations* (page 329). For information on how applications might use indexes, see *Indexing Strategies* (page 321).

CHAPTER 22

Index Concepts

22.1 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 329) document. For strategies and practical approaches, see the *Indexing Strategies* (page 321) document.

22.1.1 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.
- You can create indexes on a single field or on multiple fields using a *compound index* (page 311).
- Indexes enhance query performance, often dramatically. However, each index also incurs some overhead for every write operation. Consider the queries, the frequency of these queries, the size of your working set, the insert load, and your application's requirements as you create indexes in your MongoDB environment.
- All MongoDB indexes use a B-tree data structure. MongoDB can use this representation of the data to optimize query responses.
- Every query, including update operations, uses one and only one index. The *query optimizer* (page 48) 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()` (page 985) method.
- An index "covers" a query if:
  - all the fields in the *query* (page 42) are part of that index, **and**
  - all the fields returned in the documents that match the query are in the same index.

  When an index covers a query, the server can both match the *query conditions* (page 42) **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query. Querying the index can be faster than querying the documents outside of the index.

  See *Create Indexes that Support Covered Queries* (page 322) for more information.
- Using queries with good index coverage reduces the number of full documents that MongoDB needs to store in memory, thus maximizing database performance and throughput.

- If an update does not change the size of a document or cause the document to outgrow its allocated area, then MongoDB will update an index *only if* the indexed fields have changed. This improves performance. Note that if the document has grown and must move, all index keys must then update.

22.1.2 Index Types

This section enumerates the types of indexes available in MongoDB. For all collections, MongoDB creates the default *_id index* (page 310). You can create additional indexes with the `ensureIndex()` (page 949) method on any single field or *sequence of fields* (page 311) within any document or *sub-document* (page 310). MongoDB also supports indexes of arrays, called *multi-key indexes* (page 312).

_id Index

The `_id` index is a *unique index* (page 314) [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*. If you use `insert()` (page 961) and do not specify an *ObjectID*, MongoDB will create an `_id` field and generate an *ObjectID*. *ObjectIds* are 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 while scanning fewer whole documents.

In general, you should create indexes that support your primary, common, and user-facing queries. Doing so requires MongoDB to scan the fewest number of documents possible.

In the `mongo` (page 1066) shell, you can create an index by calling the `ensureIndex()` (page 949) method. Arguments to `ensureIndex()` (page 949) resemble the following:

```
{ "field": 1 }
{ "product.quantity": 1 }
{ "product": 1, "quantity": 1 }
```

For each field in the index 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* (page 311)) the sequence of fields is important.

Indexes on Sub-documents

You can create indexes on fields that hold sub-documents as in the following example:

[1] Although the index on `_id` *is* unique, the `getIndexes()` (page 956) method will *not* print `unique: true` in the `mongo` (page 1066) 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 1194) for more information.

Example

Given the following document in the `factories` collection:

```
{ "_id": ObjectId(...), metro: { city: "New York", state: "NY" } } )
```

You can create an index on the `metro` key. The following queries would then use that index, and both would return the above document:

```
db.factories.find( { metro: { city: "New York", state: "NY" } } );

db.factories.find( { metro: { $gte : { city: "New York" } } } );
```

The second query returns the document because `{ city: "New York" }` is less than `{ city: "New York", state: "NY" }` The order of comparison is in ascending key order in the order the keys occur in the *BSON* document.

Indexes on Embedded Fields

You can create indexes on fields in sub-documents, just as you can index top-level fields in documents. [3] These indexes allow you to use a "dot notation," to introspect into sub-documents.

Consider a collection named `people` that holds documents that resemble the following example document:

```
{"_id": ObjectId(...)
 "name": "John Doe"
 "address": {
        "street": "Main"
        "zipcode": 53511
        "state": "WI"
        }
}
```

You can create an index on the `address.zipcode` field, using the following specification:

```
db.people.ensureIndex( { "address.zipcode": 1 } )
```

Compound Indexes

MongoDB supports "compound indexes," where a single index structure holds references to multiple fields [4] within a collection's documents. Consider a collection named `products` that holds documents that resemble the following document:

```
{
 "_id": ObjectId(...)
 "item": "Banana"
 "category": ["food", "produce", "grocery"]
 "location": "4th Street Store"
 "stock": 4
 "type": cases
 "arrival": Date(...)
}
```

[3] *Indexes on Sub-documents* (page 310), by contrast allow you to index fields that hold documents, including the full content, up to the maximum `Index Size` (page 1140) of the sub-document in the index.

[4] MongoDB imposes a `limit of 31 fields for any compound index` (page 1140).

If most applications queries include the `item` field and 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, "location": 1, "stock": 1 } )
```

Compound indexes support queries on any prefix of the fields in the index. [5] For example, MongoDB can use the above index to support queries that select the `item` field and to support queries that select the `item` field **and** the `location` field. The index, however, would not support queries that select the following:

- only the `location` field
- only the `stock` field
- only the `location` and `stock` fields
- only the `item` and `stock` fields

Important: You may not create compound indexes that have `hashed` index fields. You will receive an error if you attempt to create a compound index that includes *a hashed index* (page 332).

When creating an index, the number associated with a key specifies the direction of the index. The options are `1` (ascending) and `-1` (descending). Direction doesn't matter for single key indexes or for random access retrieval but is important if you are doing sort queries on compound indexes.

The order of fields in a compound index is very important. In the previous example, the index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by the values of `location`, and then sorted by values of the `stock` field.

Indexes with Ascending and Descending Keys

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* (page 311), 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 311), the specification would resemble the following prototype:

```
db.products.ensureIndex( { "fieldA": 1, "fieldB": -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 Indexes

If you index a field that contains an array, MongoDB indexes each value in the array separately, in a "multikey index."

Example

Given the following document:

[5] Index prefixes are the beginning subset of fields. For example, given the index `{ a: 1, b: 1, c: 1 }` both `{ a: 1 }` and `{ a: 1, b: 1 }` are prefixes of the index.

```
{ "_id" : ObjectId("..."),
  "name" : "Warm Weather",
  "author" : "Steve",
  "tags" : [ "weather", "hot", "record", "april" ] }
```

Then an index on the `tags` field would be a multikey index and would include these separate entries:

```
{ tags: "weather" }
{ tags: "hot" }
{ tags: "record" }
{ tags: "april" }
```

Queries could use the multikey index to return queries for any of the above values.

Note: For `hashed` indexes, MongoDB collapses sub-documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes. For fields that hold sub-documents, you cannot use the index to support queries that introspect the sub-document.

You can use multikey indexes to index fields within objects embedded in arrays, as in the following example:

Example

Consider a `feedback` collection with documents in the following form:

```
{
 "_id": ObjectId(...)
 "title": "Grocery Quality"
 "comments": [
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the cheddar selection." },
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the mustard selection." },
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the olive selection." }
 ]
}
```

An index on the `comments.text` field would be a multikey index and would add items to the index for all of the sub-documents in the array.

With an index, such as `{ comments.text: 1 }`, consider the following query:

```
db.feedback.find( { "comments.text": "Please expand the olive selection." } )
```

This would select the document, that contains the following document in the `comments.text` array:

```
{ author_id: ObjectId(...)
  date: Date(...)
  text: "Please expand the olive selection." }
```

Compound Multikey Indexes May Only Include One Array Field

While you can create multikey *compound indexes* (page 311), at most one field in a compound index may hold an array. For example, given an index on `{ a: 1, b: 1 }`, the following documents are permissible:

```
{a: [1, 2], b: 1}
```

```
{a: 1, b: [1, 2]}
```

However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the `{a: 1, b: 1 }` index:

```
{a: [1, 2], b: [1, 2]}
```

If you attempt to insert a such a document, MongoDB will reject the insertion, and produce an error that says `cannot index parallel arrays`. MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field. To create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` (page 1066) shell:

```
db.addresses.ensureIndex( { "user_id": 1 }, { unique: true } )
```

By default, `unique` is `false` on MongoDB indexes.

If you use the unique constraint on a *compound index* (page 311) 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 314) to filter these null values from the unique index.

You may not specify a unique constraint on a *hashed index* (page 315).

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field. [6] Any document that is missing the field is not indexed. The index is "sparse" because of the missing documents when values are missing.

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` (page 1066) shell:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

By default, `sparse` is `false` on MongoDB indexes.

> **Warning:** Using these indexes will sometimes result in incomplete results when filtering or sorting results, because sparse indexes are not complete for all documents in a collection.

Note: Do not confuse sparse indexes in MongoDB with block-level 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 314) option so that `mongod` (page 1049) will reject documents that have duplicate values for a field, but that ignore documents that do not have the key.

[6] All documents that have the indexed field *are* indexed in a sparse index, even if that field stores a null value in some documents.

Hashed Index

New in version 2.4.

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses sub-documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

MongoDB can use the `hashed` index to support equality queries, but `hashed` indexes do not support range queries.

You may not create compound indexes that have `hashed` index fields or specify a unique constraint on a `hashed` index; however, you can create both a `hashed` index and an ascending/descending (i.e. non-hashed) index on the same field: MongoDB will use the scalar index for range queries.

> **Warning:** MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be consistently converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

Create a `hashed` index using an operation that resembles the following:

```
db.active.ensureIndex( { a: "hashed" } )
```

This operation creates a hashed index for the `active` collection on the `a` field.

22.1.3 Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index (1 or -1).

Example

Issue the following command to create an index on `item` and `quantity`:

```
db.products.ensureIndex( { item: 1, quantity: -1 } )
```

The resulting index is named: `item_1_quantity_-1`.

Optionally, you can specify a name for an index instead of using the default name.

Example

Issue the following command to create an index on `item` and `quantity` and specify `inventory` as the index name:

```
db.products.ensureIndex( { item: 1, quantity: -1 } , {name: "inventory"} )
```

The resulting index is named: `inventory`.

To view the name of an index, use the `getIndexes()` (page 956) method.

22.1.4 Index Creation Options

You specify index creation options in the second argument in `ensureIndex()` (page 949).

The options *sparse* (page 314), *unique* (page 314), and *TTL* (page 317) affect the kind of index that MongoDB creates. This section addresses, *background construction* (page 316) and *duplicate dropping* (page 317), which 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 can take a long time to complete. To resolve this issue, the background option can allow you to continue to use your `mongod` (page 1049) instance during the index build.

For example, to create an index in the background of the `zipcode` field of the `people` collection you would issue the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

By default, `background` is `false` for building MongoDB indexes.

You can combine the background option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true, sparse: true } )
```

Be aware of the following behaviors with background index construction:

- A `mongod` (page 1049) instance can build more than one index in the background concurrently.

  Changed in version 2.4: Before 2.4, a `mongod` (page 1049) instance could only build one background index per database at a time.

  Changed in version 2.2: Before 2.2, a single `mongod` (page 1049) 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` (page 1066) shell session or connection where you are creating the index will block until the index build is complete. Open another connection or `mongo` (page 1066) 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.

- If your application includes `ensureIndex()` (page 949) operations, and an index *doesn't* exist for other operational concerns, building the index can have a severe impact on the performance of the database.

  Make sure that your application checks for the indexes at start up using the `getIndexes()` (page 956) method or the equivalent method for your driver and terminates if the proper indexes do not exist. Always build indexes in production instances using separate application code, during designated maintenance windows.

Building Indexes on Secondaries

Background index operations on a *replica set primary* become foreground indexing operations on secondary members of the set. All indexing operations on secondaries block replication.

To build large indexes on secondaries the best approach is to restart one secondary at a time in *standalone* mode and build the index. After building the index, restart as a member of the replica set, allow it to catch up with the other members of the set, and then build the index on the next secondary. When all the secondaries have the new index, step down the primary, restart it as a standalone, and build the index on the former primary.

Remember, the amount of time required to build the index on a secondary node must be within the window of the *oplog*, so that the secondary can catch up with the primary.

See *Build Indexes on Replica Sets* (page 333) 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.

See *Build Indexes on Replica Sets* (page 333) for a complete procedure for rebuilding indexes on secondaries.

Note: If MongoDB is building an index in the background, you cannot perform other administrative operations involving that collection, including `repairDatabase` (page 895), drop that collection (i.e. `db.collection.drop()` (page 948),) and `compact` (page 890). These operations will return an error during background index builds.

Queries will not use these indexes until the index build is complete.

Drop Duplicates

MongoDB cannot create a *unique index* (page 314) 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 314) option, this index will only include documents in the index that have the value, and the documents without the field will remain in the database.

To create a unique index that drops duplicates on the `username` field of the `accounts` collection, use a command in the following form:

```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

> **Warning:** Specifying `{ dropDups: true }` will delete data from your database. Use with extreme caution.

By default, `dropDups` is `false`.

22.1.5 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* (page 311) 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 *every 60 seconds*. 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 (page 1049) instance.

In all other respects, TTL indexes are normal indexes, and if appropriate, MongoDB can use these indexes to fulfill arbitrary queries.

See

Expire Data from Collections by Setting TTL (page 599)

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` (page 801), and the database command `geoNear` (page 850) for more information on accessing geospatial data.

Geohaystack Indexes

In addition to conventional *geospatial indexes* (page 318), 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.

`text` Indexes

New in version 2.4.

MongoDB provides `text` indexes to support the search of string content in documents of a collection. `text` indexes are case-insensitive and can include any field that contains string data. `text` indexes drop language-specific stop words (e.g. in English, "the," "an," "a," "and," etc.) and uses simple language-specific suffix stemming. See *Text Search Languages* (page 860) for the supported languages.

You can only access the `text` index with the `text` (page 856) command.

See *Text Search* (page 353) for more information.

22.1.6 Index Behaviors

Limitations

- A collection may have no more than *64 indexes* (page 1140).
- Index keys can be no larger than *1024 bytes* (page 1140).

  Documents with fields that have values greater than this size cannot be indexed.

  To query for documents that were too large to index, you can use a command similar to the following:

  ```
  db.records.find({<key>: <value too large to index>}).hint({$natural: 1})
  ```

- The name of an index, including the *namespace* must be shorter than *128 characters* (page 1140).
- 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.
- For queries with the `$or` (page 790) operator, each clause of an `$or` (page 790) query executes in parallel, and can each use a different index.
- For queries that use the `sort()` (page 991) method and use the `$or` (page 790) operator, the query **cannot** use the indexes on the `$or` (page 790) fields.
- `2d` *geospatial queries* (page 351) do not support queries that use the `$or` (page 790) operator.

Consider Insert Throughput

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 to complement your queries. Always have a good reason for adding a new index, and be sure to benchmark alternative strategies.

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 MongoDB only modifies 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 savings in read-time.
- If you have many related indexes on a collection that receives a high volume of write operations, 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 326), the speed improvements for query operations may not offset the costs of maintaining an index. For more information see *Create Queries that Ensure Selectivity* (page 326).

CHAPTER 23

Indexing Strategies for Applications

23.1 Indexing 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 to see which configurations perform best.

MongoDB can only use *one* index to support any given operation. However, each clause of an `$or` (page 790) query may use a different index.

The following documents introduce indexing strategies:

23.1.1 Create Indexes to Support Your Queries

An index supports a query when the index contains all the fields scanned by the query. The query scans the index and not the collection. Creating indexes that supports queries results in greatly increased query performance.

This document describes strategies for creating indexes that support queries.

Create a Single-Key Index if All Queries Use the Same, Single Key

If you only ever query on a single key in a given collection, then you need to create just one single-key index for that collection. For example, you might create an index on `category` in the `product` collection:

```
db.products.ensureIndex( { "category": 1 } )
```

Create Compound Indexes to Support Several Different Queries

If you sometimes query on only one key and at other times query on that key combined with a second key, then creating a compound index is more efficient than creating a single-key index. MongoDB will use the compound index for both queries. For example, you might create an index on both `category` and `item`.

```
db.products.ensureIndex( { "category": 1, "item": 1 } )
```

This allows you both options. You can query on just `category`, and you also can query on `category` combined with `item`. A single *compound index* (page 311) on multiple fields can support all the queries that search a "prefix" subset of those fields.

Note: With the exception of queries that use the `$or` (page 790) operator, a query does not use multiple indexes. A query uses only one index.

Example

The following index on a collection:

```
{ x: 1, y: 1, z: 1 }
```

Can support queries that the following indexes support:

```
{ x: 1 }
{ x: 1, y: 1 }
```

There are some situations where the prefix indexes may offer better query performance: for example if `z` is a large array.

The `{ x: 1, y: 1, z: 1 }` index can also support many of the same queries as the following index:

```
{ x: 1, z: 1 }
```

Also, `{ x: 1, z: 1 }` has an additional use. Given the following query:

```
db.collection.find( { x: 5 } ).sort( { z: 1} )
```

The `{ x: 1, z: 1 }` index supports both the query and the sort operation, while the `{ x: 1, y: 1, z: 1 }` index only supports the query. For more information on sorting, see *Use Indexes to Sort Query Results* (page 323).

Create Indexes that Support Covered Queries

A covered query is a query in which:

- all the fields in the *query* (page 42) are part of an index, **and**
- all the fields returned in the results are in the same index.

Because the index "covers" the query, MongoDB can both match the *query conditions* (page 42) **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query.

Querying *only* the index can be much faster than querying documents outside of the index. Index keys are typically smaller than the documents they catalog, and indexes are typically available in RAM or located sequentially on disk.

MongoDB automatically uses an index that covers a query when possible. To ensure that an index can *cover* a query, create an index that includes all the fields listed in the *query document* (page 42) and in the query result. You can specify the fields to return in the query results with a *projection* (page 45) document. By default, MongoDB includes the _id field in the query result. So, if the index does **not** include the _id field, then you must exclude the _id field (i.e. _id: 0) from the query results.

Example

Given collection users with an index on the fields user and status, as created by the following option:

```
db.users.ensureIndex( { status: 1, user: 1 } )
```

Then, this index will cover the following query which selects on the status field and returns only the user field:

```
db.users.find( { status: "A" }, { user: 1, _id: 0 } )
```

In the operation, the projection document explicitly specifies _id: 0 to exclude the _id field from the result since the index is only on the status and the user fields.

If the projection document does not specify the exclusion of the _id field, the query returns the _id field. The following query is **not** covered by the index on the status and the user fields because with the projection document { user: 1 }, the query returns both the user field and the _id field:

```
db.users.find( { status: "A" }, { user: 1 } )
```

An index **cannot** cover a query if:

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* (page 312) index and cannot support a covered query.
- any of the indexed fields are fields in subdocuments. To index fields in subdocuments, use *dot notation*. For example, consider a collection users with documents of the following form:

  ```
  { _id: 1, user: { login: "tester" } }
  ```

  The collection has the following indexes:

  ```
  { user: 1 }
  ```

  ```
  { "user.login": 1 }
  ```

  The { user: 1 } index covers the following query:

  ```
  db.users.find( { user: { login: "tester" } }, { user: 1, _id: 0 } )
  ```

  However, the { "user.login": 1 } index does **not** cover the following query:

  ```
  db.users.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
  ```

  The query, however, does use the { "user.login": 1 } index to find matching documents.

To determine whether a query is a covered query, use the explain() (page 979) method. If the explain() (page 979) output displays true for the indexOnly (page 983) field, the query is covered by an index, and MongoDB queries only that index to match the query **and** return the results.

For more information see *Measure Index Use* (page 337).

23.1.2 Use Indexes to Sort Query Results

In MongoDB sort operations that sort documents based on an indexed field provide the greatest performance. Indexes in MongoDB, as in other databases, have an order: as a result, using an index to access documents returns in the same order as the index.

To sort on multiple fields, create a *compound index* (page 311). With compound indexes, the results can be in the sorted order of either the full index or an index prefix. An index prefix is a subset of a compound index; the subset consists of one or more fields at the start of the index, in order. For example, given an index `{ a:1, b: 1, c: 1, d: 1 }`, the following subsets are index prefixes:

```
{ a: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 1, c: 1 }
```

For more information on sorting by index prefixes, see *Sort Subset Starts at the Index Beginning* (page 324).

If the query includes **equality** match conditions on an index prefix, you can sort on a subset of the index that starts after or overlaps with the prefix. For example, given an index `{ a: 1, b: 1, c: 1, d: 1 }`, if the query condition includes equality match conditions on `a` and `b`, you can specify a sort on the subsets `{ c: 1 }` or `{ c: 1, d: 1 }`:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1, d: 1 } )
```

In these operations, the equality match and the sort documents together cover the index prefixes `{ a: 1, b: 1, c: 1 }` and `{ a: 1, b: 1, c: 1, d: 1 }` respectively.

You can also specify a sort order that includes the prefix; however, since the query condition specifies equality matches on these fields, they are constant in the resulting documents and do not contribute to the sort order:

```
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1 } )
db.collection.find( { a: 5, b: 3 } ).sort( { a: 1, b: 1, c: 1, d: 1 } )
```

For more information on sorting by index subsets that are not prefixes, see *Sort Subset Does Not Start at the Index Beginning* (page 325).

Note: For in-memory sorts that do not use an index, the `sort()` (page 991) operation is significantly slower. The `sort()` (page 991) operation will abort when it uses 32 megabytes of memory.

Sort With a Subset of Compound Index

If the sort document contains a subset of the compound index fields, the subset can determine whether MongoDB can use the index efficiently to both retrieve and sort the query results. If MongoDB can efficiently use the index to both retrieve and sort the query results, the output from the `explain()` (page 979) will display `scanAndOrder` (page 983) as `false` or `0`. If MongoDB can only use the index for retrieving documents that meet the query criteria, MongoDB must manually sort the resulting documents without the use of the index. For in-memory sort operations, `explain()` (page 979) will display `scanAndOrder` (page 983) as `true` or `1`.

Sort Subset Starts at the Index Beginning

If the sort document is a subset of a compound index and starts from the beginning of the index, MongoDB can use the index to both retrieve and sort the query results.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following operations include a sort with a subset of the index. Because the sort subset starts at beginning of the index, the operations can use the index for both the query retrieval and sort:

```
db.collection.find().sort( { a:1 } )
db.collection.find().sort( { a:1, b:1 } )
db.collection.find().sort( { a:1, b:1, c:1 } )

db.collection.find( { a: 4 } ).sort( { a: 1, b: 1 } )
db.collection.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )

db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

The last two operations include query conditions on the field `b` but does not include a query condition on the field `a`:

```
db.collection.find( { b: 5 } ).sort( { a: 1, b: 1 } )
db.collection.find( { b: { $gt:5 }, c: { $gt: 1 } } ).sort( { a: 1, b: 1 } )
```

Consider the case where the collection has the index `{ b: 1 }` in addition to the `{ a: 1, b: 1, c: 1, d: 1 }` index. Because of the query condition on `b`, it is not immediately obvious which index MongoDB may select as the "best" index. To explicitly specify the index to use, see `hint()` (page 985).

Sort Subset Does Not Start at the Index Beginning

The sort document can be a subset of a compound index that does **not** start from the beginning of the index. For instance, `{ c: 1 }` is a subset of the index `{ a: 1, b: 1, c: 1, d: 1 }` that omits the preceding index fields `a` and `b`. MongoDB can use the index efficiently **if** the the query document includes all the preceding fields of the index, in this case `a` and `b`, in **equality** conditions. In other words, the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index.

For example, the collection `collection` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

Then following operations can use the index efficiently:

```
db.collection.find( { a: 5 } ).sort( { b: 1, c: 1 } )
db.collection.find( { a: 5, c: 4, b: 3 } ).sort( { d: 1 } )
```

- In the first operation, the query document `{ a: 5 }` with the sort document `{ b: 1, c: 1 }` cover the prefix `{ a:1 , b: 1, c: 1 }` of the index.
- In the second operation, the query document `{ a: 5, c: 4, b: 3 }` with the sort document `{ d: 1 }` covers the full index.

Only the index fields preceding the sort subset must have the equality conditions in the query document. The other index fields may have other conditions. The following operations can efficiently use the index since the equality conditions in the query document and the subset in the sort document **contiguously** cover a prefix of the index:

```
db.collection.find( { a: 5, b: 3 } ).sort( { c: 1 } )
db.collection.find( { a: 5, b: 3, c: { $lt: 4 } } ).sort( { c: 1 } )
```

The following operations specify a sort document of `{ c: 1 }`, but the query documents do not contain equality matches on the **preceding** index fields `a` and `b`:

```
db.collection.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
db.collection.find( { c: 5 } ).sort( { c: 1 } )
```

These operations **will not** efficiently use the index `{ a: 1, b: 1, c: 1, d: 1 }` and may not even use the index to retrieve the documents.

23.1.3 Ensure Indexes Fit RAM

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` (page 973) helper, which returns data in bytes:

```
> db.collection.totalIndexSize()
4294976499
```

The above example shows an index size of almost 4.3 gigabytes. To ensure this index fits in RAM, you must not only have more than that much RAM available but also must have RAM available for the rest of the *working set*. Also remember:

If you have and use multiple collections, you must consider the size of all indexes on all collections. The indexes and the working set must be able to fit in memory at the same time.

There are some limited cases where indexes do not need to fit in memory. See *Indexes that Hold Only Recent Values in RAM* (page 326).

See also:

`collStats` (page 900) and `db.collection.stats()` (page 973)

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit *entirely* into RAM in all cases. If the value of the indexed field increments with every insert, and most queries select recently added documents; then MongoDB only needs to keep the parts of the index that hold the most recent or "right-most" values in RAM. This allows for efficient index use for read and write operations and minimize the amount of RAM required to support the index.

23.1.4 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 311) that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Another option, again depending on your use case, might be to use separate collections, one for each status.

Example

Consider an index `{ a : 1 }` (i.e. an index on the key `a` sorted in ascending order) on a collection where `a` has three values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 1, b: "cd" }
{ _id: ObjectId(), a: 1, b: "ef" }
```

```
{ _id: ObjectId(), a: 2, b: "jk" }
{ _id: ObjectId(), a: 2, b: "lm" }
{ _id: ObjectId(), a: 2, b: "no" }
{ _id: ObjectId(), a: 3, b: "pq" }
{ _id: ObjectId(), a: 3, b: "rs" }
{ _id: ObjectId(), a: 3, b: "tv" }
```

If you query for `{ a: 2, b: "no" }` MongoDB must scan 3 *documents* in the collection to return the one matching result. Similarly, a query for `{ a: { $gt: 1}, b: "tv" }` must scan 6 documents, also to return one result.

Consider the same index on a collection where `a` has *nine* values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 2, b: "cd" }
{ _id: ObjectId(), a: 3, b: "ef" }
{ _id: ObjectId(), a: 4, b: "jk" }
{ _id: ObjectId(), a: 5, b: "lm" }
{ _id: ObjectId(), a: 6, b: "no" }
{ _id: ObjectId(), a: 7, b: "pq" }
{ _id: ObjectId(), a: 8, b: "rs" }
{ _id: ObjectId(), a: 9, b: "tv" }
```

If you query for `{ a: 2, b: "cd" }`, MongoDB must scan only one document to fulfill the query. The index and query are more selective because the values of `a` are evenly distributed *and* the query can select a specific document using the index.

However, although the index on `a` is more selective, a query such as `{ a: { $gt: 5 }, b: "tv" }` would still need to scan 4 documents.

If overall selectivity is low, and if MongoDB must read a number of documents to return results, then some queries may perform faster without indexes. To determine performance, see *Measure Index Use* (page 337).

For a conceptual introduction to indexes in MongoDB see *Indexing Overview* (page 309).

CHAPTER 24

Index Tutorials

24.1 Indexing Operations

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

The documents in this section outline specific tasks related to building and maintaining indexes for data in MongoDB collections. For a conceptual overview of MongoDB indexing, see the *Indexing Overview* (page 309) document. For strategies and practical approaches, see the *Indexing Strategies* (page 321) document.

24.1.1 Create an Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB creates an index on the `_id` field of every collection by default, but allows users to create indexes for any collection using on any field in a *document*.

This tutorial describes how to create an index on a single field. MongoDB also supports *compound indexes* (page 311), which are indexes on multiple fields. See *Create a Compound Index* (page 330) for instructions on building compound indexes.

Build a Foreground Index on a Single Field

To create an index, use `db.collection.ensureIndex()` (page 949) or a similar method from your driver. For example the following creates an index on the `phone-number` field of the `people` collection:

```
db.people.ensureIndex( { "phone-number": 1 } )
```

`ensureIndex()` (page 949) only creates an index if an index of the same specification does not already exist.

All indexes support and optimize the performance for queries that select on this field. For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Examples

If you create an index on the `user_id` field in the `records`, this index is, the index will support the following query:

```
db.records.find( { user_id: 2 } )
```

However, the following query, on the `profile_url` field is not supported by this index:

```
db.records.find( { profile_url: 2 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in *Background Construction* (page 316). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 333) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 333).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

See also:

Create a Compound Index (page 330), *Indexing Operations* (page 329) and *Indexing Overview* (page 309) for more information.

24.1.2 Create a Compound Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. MongoDB supports indexes that include content on a single field, as well as *compound indexes* (page 311) that include content from multiple fields. Continue reading for instructions and examples of building a compound index.

Build a Compound Index

To create a *compound index* (page 311) use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

Example

The following operation will create an index on the `item`, `category`, and `price` fields of the `products` collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

Additional Considerations

If your collection holds a large amount of data, and your application needs to be able to access the data while building the index, consider building the index in the background, as described in *Background Construction* (page 316). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 333) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 333).

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

See also:

Create an Index (page 329), *Indexing Operations* (page 329) and *Indexing Overview* (page 309) for more information.

24.1.3 Create a Unique Index

MongoDB allows you to specify a *unique constraint* (page 314) on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields. Additionally, if you want to create an index on a collection that has existing data that might have duplicate values for the indexed field, you may chose combine unique enforcement with *duplicate dropping* (page 317).

Unique Indexes

To create a *unique indexes* (page 314), 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 310) 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 311), as in the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1 }, { unique: true } )
```

These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Drop Duplicates

To force the creation of a *unique index* (page 314) 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()` (page 949):

```
db.collection.ensureIndex( { a: 1 }, { unique: true, dropDups: true } )
```

See the full documentation of *duplicate dropping* (page 317) for more information.

> **Warning:** Specifying `{ dropDups: true }` may delete data from your database. Use with extreme caution.

Refer to the `ensureIndex()` (page 949) documentation for additional index creation options.

24.1.4 Create a Sparse Index

Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings. See *Sparse Indexes* (page 314) for more information about sparse indexes and their use.

See also:

Indexing Overview (page 309) and *Indexing Operations* (page 329) for more information.

Prototype

To create a *sparse index* (page 314) on a field, use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

Example

The following operation, creates a sparse index on the `users` collection that *only* includes a document in the index if the `twitter_name` field exists in a document.

```
db.users.ensureIndex( { twitter_name: 1 }, { sparse: true } )
```

The index excludes all documents that do not include the `twitter_name` field.

Considerations

Note: Sparse indexes can affect the results returned by the query, particularly with respect to sorts on fields *not* included in the index. See the *sparse index* (page 314) section for more information.

24.1.5 Create a Hashed Index

New in version 2.4.

Hashed indexes (page 315) compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

See

Hashed Shard Keys (page 502) for more information about hashed indexes in sharded clusters, as well as *Indexing Overview* (page 309) and *Indexing Operations* (page 329) for more information about indexes.

Procedure

To create a *hashed index* (page 315), specify `hashed` as the value of the index key, as in the following prototype:

Example

```
db.collection.ensureIndex( { a: "hashed" } )
```

Considerations

MongoDB supports `hashed` indexes of any single field. The hashing function collapses sub-documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

You may not create compound indexes that have `hashed` index fields.

24.1.6 Build Indexes on Replica Sets

Background index creation operations (page 316) 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 1061) will send `ensureIndex()` (page 949) to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.

To minimize the impact of building an index on your replica set, use the following procedure to build indexes on secondaries:

See

Indexing Operations (page 329) and *Indexing Overview* (page 309) for more information.

Considerations

Warning: Ensure that your *oplog* is large enough to permit the indexing or re-indexing operation to complete without falling too far behind to catch up. See the "*oplog sizing* (page 405)" documentation for additional information.

Note: This procedure *does* take one member out of the replica set at a time. However, this procedure will only affect one member of the set at a time rather than *all* secondaries at the same time.

Procedure

Note: If you need to build an index in a *sharded cluster*, repeat the following procedure for each replica set that provides each *shard*.

Stop One Secondary

Stop the `mongod` (page 1049) process on one secondary. Restart the `mongod` (page 1049) process *without* the `--replSet` option and running on a different port. [1] This instance is now in "standalone" mode.

For example, if your `mongod` (page 1049) *normally* runs with on the default port of `27017` with the `--replSet` option you would use the following invocation:

```
mongod --port 47017
```

Build the Index

Create the new index using the `ensureIndex()` (page 949) in the `mongo` (page 1066) shell, or comparable method in your driver. This operation will create or rebuild the index on this `mongod` (page 1049) instance

For example, to create an ascending index on the `username` field of the `records` collection, use the following `mongo` (page 1066) shell operation:

[1] By running the `mongod` (page 1049) 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.

```
db.records.ensureIndex( { username: 1 } )
```

See also:

Create an Index (page 329) and *Create a Compound Index* (page 330) for more information.

Restart the Program `mongod`

When the index build completes, start the `mongod` (page 1049) instance with the `--replSet` option on its usual port:

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

Modify the port number (e.g. `27017`) or the replica set name (e.g. `rs0`) as needed.

Allow replication to catch up on this member.

Build Indexes on all Secondaries

For each secondary in the set, build an index according to the following steps:

1. *Stop One Secondary* (page 333)
2. *Build the Index* (page 333)
3. *Restart the Program mongod* (page 334)

Build the Index on the Primary

Finally, to build the index on the *primary*, begin by stepping down the primary. Use the `rs.stepDown()` (page 1018) method in the `mongo` (page 1066) shell to cause the current primary to become a secondary graceful and allow the set to elect another member as primary.

Then repeat the index building procedure, listed below, to build the index on the primary:

1. *Stop One Secondary* (page 333)
2. *Build the Index* (page 333)
3. *Restart the Program mongod* (page 334)

24.1.7 Build Indexes in the Background

By default, MongoDB builds indexes in the foreground, which means that these indexes block all other read and write operations to the database while the index builds. *Background index construction* (page 316) allows read and write operations to continue while building the index; however, these index builds take longer to complete and result in a larger index.

After the index finishes building, MongoDB treats indexes built in the background the same as any other index.

See also:

Indexing Overview (page 309) and *Indexing Operations* (page 329) for more information.

Procedure

To create an index in the background, add the `background` argument to the `ensureIndex()` (page 949) operation, as in the following index:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on *background index construction* (page 316) for more information about these indexes and their implications.

24.1.8 Remove Indexes

To remove an index from a collection use the `dropIndex()` method and the following procedure. If you simply need to rebuild indexes you can use the process described in the *Rebuild Indexes* (page 335) document.

See also:

Indexing Operations (page 329) and *Indexing Overview* (page 309) for more information about indexes and indexing operations in MongoDB.

Operations

To remove an index, use the `db.collection.dropIndex()` method, as in the following example:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

This will remove the index on the `"tax-id"` field in the `accounts` collection. The shell provides the following document after completing the operation:

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index. You can also use the `db.collection.dropIndexes()` (page 949) to remove *all* indexes, except for the *_id index* (page 310) from a collection.

These shell helpers provide wrappers around the `dropIndexes` (page 888) *database command*. Your *client library* (page 575) may have a different or additional interface for these operations.

24.1.9 Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` (page 970) method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the *_id index* (page 310), and then rebuilds all indexes.

See also:

Indexing Overview (page 309) and *Indexing Operations* (page 329).

Process

The operation takes the following form:

```
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        "indexes" : [
                {
                        "key" : {
                                "_id" : 1,
                                "tax-id" : 1
                        },
                        "ns" : "records.accounts",
                        "name" : "_id_"
                }
        ],
        "ok" : 1
}
```

This shell helper provides a wrapper around the `reIndex` (page 894) *database command*. Your *client library* (page 575) may have a different or additional interface for this operation.

Additional Considerations

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 333).

24.1.10 Monitor and Manage In Progress Index Creation and Building

To see the status of the indexing processes, you can use the `db.currentOp()` (page 998) method in the `mongo` (page 1066) shell. The value of the `query` field and the `msg` field will indicate if the operation is an index build. The `msg` field also indicates the percent of the build that is complete.

To terminate an ongoing index build, use the `db.killOp()` (page 1009) method in the `mongo` (page 1066) shell.

For more information see `db.currentOp()` (page 998).

Changed in version 2.4: Before MongoDB 2.4, you could *only* terminate *background* index builds. After 2.4, you can terminate any index build, including foreground index builds.

24.1.11 Return a List of All Indexes

When performing maintenance you may want to check which indexes exist on a collection. Every index on a collection has a corresponding *document* in the `system.indexes` (page 1133) collection, and you can use standard queries (i.e. `find()` (page 951)) to list the indexes, or in the `mongo` (page 1066) shell, the `getIndexes()` (page 956) method to return a list of the indexes on a collection, as in the following examples.

See also:

Indexing Overview (page 309) and *Indexing Operations* (page 329) for more information about indexes in MongoDB and common index management operations.

List all Indexes on a Collection

To return a list of all indexes on a collection, use the `db.collection.getIndexes()` (page 956) method or a similar method for your driver.

For example, to view all indexes on the `people` collection:

```
db.people.getIndexes()
```

List all Indexes for a Database

To return a list of all indexes on all collections in a database, use the following operation in the `mongo` (page 1066) shell:

```
db.system.indexes.find()
```

See `system.indexes` (page 1133) for more information about these documents.

24.1.12 Measure Index Use

Synopsis

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides a number of tools that allow you to study query operations and observe index use for your database.

See also:

Indexing Overview (page 309), *Indexing Strategies* (page 321), and *Indexing Operations* (page 329) for more information.

Operations

Return Query Plan with explain()

Append the `explain()` (page 979) method to any cursor (e.g. query) to return a document with statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Control Index Use with hint()

Append the `hint()` (page 985) 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()` (page 985) and `explain()` (page 979) in conjunction with each other to compare the effectiveness of a specific index. Specify the `$natural` operator to the `hint()` (page 985) method to prevent MongoDB from using *any* index:

```
db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } } ).hint( { $natural: 1 } )
```

Instance Index Use Reporting

MongoDB provides a number of metrics of index use and operation that you may want to consider when analyzing index use for your database:

- In the output of `serverStatus` (page 919):

  - `indexCounters` (page 925)
  - `scanned` (page 933)
  - `scanAndOrder` (page 933)
- In the output of `collStats` (page 900):
  - `totalIndexSize` (page 902)
  - `indexSizes` (page 902)
- In the output of `dbStats` (page 904):
  - `dbStats.indexes` (page 905)
  - `dbStats.indexSize` (page 905)

24.1.13 Build Old Style Indexes

Important: Use this procedure *only* if you **must** have indexes that are compatible with a version of MongoDB earlier than 2.0.

MongoDB version 2.0 introduced the `{v:1}` index format. MongoDB versions 2.0 and later support both the `{v:1}` format and the earlier `{v:0}` format.

MongoDB versions prior to 2.0, however, support only the `{v:0}` format. If you need to roll back MongoDB to a version prior to 2.0, you must *drop* and *re-create* your indexes.

To build pre-2.0 indexes, use the `dropIndexes()` (page 949) and `ensureIndex()` (page 949) methods. You *cannot* simply reindex the collection. When you reindex on versions that only support `{v:0}` indexes, the `v` fields in the index definition still hold values of `1`, even though the indexes would now use the `{v:0}` format. If you were to upgrade again to version 2.0 or later, these indexes would not work.

Example

Suppose you rolled back from MongoDB 2.0 to MongoDB 1.8, and suppose you had the following index on the `items` collection:

```
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.items", "name" : "name_1" }
```

The `v` field tells you the index is a `{v:1}` index, which is incompatible with version 1.8.

To drop the index, issue the following command:

```
db.items.dropIndex( { name : 1 } )
```

To recreate the index as a `{v:0}` index, issue the following command:

```
db.foo.ensureIndex( { name : 1 } , { v : 0 } )
```

See also:

Index Performance Enhancements (page 1200).

CHAPTER 25

Geospatial Indexing

See *Geospatial Indexes and Queries* (page 339) for an introduction to geospatial indexing.

25.1 Geospatial Indexes and Queries

MongoDB offers a number of indexes and query mechanisms to handle geospatial information. This section introduces MongoDB's geospatial features.

25.1.1 Surfaces

Before storing your location data and writing queries, you must decide the type of surface to use to perform calculations. The type you choose affects how you store data, what type of index to build, and the syntax of your queries.

MongoDB offers two surface types:

- **Spherical**

  To calculate geometry over an Earth-like sphere, store your location data on a spherical surface and use *2dsphere* (page 344) index.

  Store your location data as GeoJSON objects with this coordinate-axis order: **longitude, latitude**. The coordinate reference system for GeoJSON uses the *WGS84* datum.

- **Flat**

  To calculate distances on a Euclidean plane, store your location data as legacy coordinate pairs and use a *2d* (page 341) index.

25.1.2 Location Data

If you choose spherical surface calculations, you store location data as

- *GeoJSON* objects (preferred).

  Queries on GeoJSON objects always calculate on a sphere. The default coordinate reference system for GeoJSON uses the *WGS84* datum.

  New in version 2.4: The storage and querying of GeoJSON objects is new in version 2.4. Prior to version 2.4, all geospatial data was stored as coordinate pairs.

  MongoDB supports the following GeoJSON objects:

– Point
– LineString
– Polygon

- *Legacy coordinate pairs*

  MongoDB supports spherical surface calculations on legacy coordinate pairs by converting the data to the GeoJSON Point type.

If you choose flat surface calculations, you can store data only as *legacy coordinate pairs*.

25.1.3 Query Operations

MongoDB's geospatial query operators let you query for:

- **Inclusion**. MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the `$geoWithin` (page 800) operator.
- **Intersection**. MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the `$geoIntersects` (page 801) operator.
- **Proximity**. MongoDB can query for the points nearest to another point. Proximity queries use the `$near` (page 801) operator. The `$near` (page 801) operator requires a `2d` or `2dsphere` index.

25.1.4 Geospatial Indexes

MongoDB provides the following geospatial index types to support the geospatial queries:

- *2dsphere* (page 344), which supports:
  – Calculations on a sphere
  – Both GeoJSON objects and legacy coordinate pairs
  – A compound index with scalar index fields (i.e. ascending or descending) as a prefix or suffix of the `2dsphere` index field

  New in version 2.4: `2dsphere` indexes are not available before version 2.4.
- *2d* (page 341), which supports:
  – Calculations using flat geometry
  – Legacy coordinate pairs (i.e., geospatial points on a flat coordinate system)
  – A compound index with only one additional field, as a suffix of the `2d` index field

25.1.5 Geospatial Indexes and Sharding

You *cannot* use a geospatial index as the *shard key* index.

You can create and maintain a geospatial index on a sharded collection if using different fields as the shard key.

Queries using `$near` (page 801) are not supported for sharded collections. Use `geoNear` (page 850) instead. You also can query for geospatial data using `$geoWithin` (page 800).

25.1.6 Additional Resources

The following pages provide complete documentation for geospatial indexes and queries:

`2d` Indexes

Use a `2d` index for data stored as points on a two-dimensional plane. The `2d` index is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

Use a `2d` index if:

- your database has legacy location data from MongoDB 2.2 or earlier, *and*
- you do not intend to store any location data as *GeoJSON* objects.

Do not use a `2d` index if your location data includes GeoJSON objects. To index on both legacy coordinate pairs *and* GeoJSON objects, use a *2dsphere* (page 344) index.

The `2d` index supports calculations on a flat, Euclidean plane. The `2d` index also supports *distance-only* calculations on a sphere, but for *geometric* calculations on a sphere, store data as GeoJSON objects and use the `2dsphere` index type.

A `2d` index can reference two fields. The first must be the location field. A `2d` compound index constructs queries that select first on the location field and second on the additional field. If the location criteria selects a large number of documents, the additional criteria only filters the result set. The additional criteria *does not* result in a more targeted query.

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2d` **or** a *2dsphere* (page 344) per collection.

Important: You cannot use a `2d` index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store Points on a 2D Plane

To store location data as legacy coordinate pairs, use either an array (preferred):

```
loc : [ <longitude> , <latitude> ]
```

Or an embedded document:

```
loc : { lng : <longitude> , lat : <latitude> }
```

Arrays are preferred as certain languages do not guarantee associative map ordering.

Whether as an array or document, if you use longitude and latitude, store coordinates in this order: **longitude, latitude**.

Create a `2d` Index

To build a geospatial `2d` index, use the `ensureIndex()` (page 949) method and specify `2d`. Use the following syntax:

```
db.<collection>.ensureIndex( { <location field> : "2d" ,
                               <additional field> : <value> } ,
                             { <index-specification options> } )
```

The `2d` index uses the following optional index-specification options:

```
{ min : <lower bound> , max : <upper bound> ,
  bits : <bit precision> }
```

Define Location Range for a `2d` Index By default, a `2d` index assumes longitude and latitude and has boundaries of -180 inclusive and 180 non-inclusive (i.e. `[ -180 , 180 ]`). If documents contain coordinate data outside of the specified range, MongoDB returns an error.

Important: The default boundaries allow applications to insert documents with invalid latitudes greater than 90 or less than -90. The behavior of geospatial queries with such invalid points is not defined.

On `2d` indexes you can change the location range.

You can build a `2d` geospatial index with a location range other than the default. Use the `min` and `max` options when creating the index. Use the following syntax:

```
db.collection.ensureIndex( { <location field> : "2d" } ,
                           { min : <lower bound> , max : <upper bound> } )
```

Define Location Precision for a `2d` Index By default, a `2d` index on legacy coordinate pairs uses 26 bits of precision, which is roughly equivalent to 2 feet or 60 centimeters of precision using the default range of -180 to 180. Precision is measured by the size in bits of the *geohash* values used to store location data. You can configure geospatial indexes with up to 32 bits of precision.

Index precision does not affect query accuracy. The actual grid coordinates are always used in the final query processing. Advantages to lower precision are a lower processing overhead for insert operations and use of less space. An advantage to higher precision is that queries scan smaller portions of the index to return results.

To configure a location precision other than the default, use the `bits` option when creating the index. Use following syntax:

```
db.<collection>.ensureIndex( {<location field> : "<index type>"} ,
                             { bits : <bit precision> } )
```

For information on the internals of geohash values, see *Calculation of Geohash Values for 2d Indexes* (page 351).

Query a `2d` Index

The following sections describe queries supported by the `2d` index. For an overview of recommended geospatial queries, see *Geospatial Query Compatibility* (page 807).

Points within a Shape Defined on a Flat Surface To select all legacy coordinate pairs found within a given shape on a flat surface, use the `$geoWithin` (page 800) operator along with a shape operator. Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                           { $box|$polygon|$center : <coordinates>
                      } } } )
```

The following queries for documents within a rectangle defined by `[ 0 , 0 ]` at the bottom left corner and by `[ 100 , 100 ]` at the top right corner.

```
db.places.find( { loc :
                  { $geoWithin :
                     { $box : [ [ 0 , 0 ] ,
                                [ 100 , 100 ] ]
                 } } } )
```

The following queries for documents that are within the circle centered on `[ -74 , 40.74 ]` and with a radius of `10`:

```
db.places.find( { loc: { $geoWithin :
                          { $center : [ [-74, 40.74 ] , 10 ]
                } } } )
```

For syntax and examples for each shape, see the following:

- `$box` (page 805)
- `$polygon` (page 806)
- `$center` (page 804) (defines a circle)

Points within a Circle Defined on a Sphere MongoDB supports rudimentary spherical queries on flat `2d` indexes for legacy reasons. In general, spherical calculations should use a `2dsphere` index, as described in *2dsphere Indexes* (page 344).

To query for legacy coordinate pairs in a "spherical cap" on a sphere, use `$geoWithin` (page 800) with the `$centerSphere` (page 805) operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distances in a 2d Index Using Spherical Geometry* (page 349).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                            { $centerSphere : [ [ <x>, <y> ] , <radius> ] }
                      } } )
```

The following example query returns all documents within a 10-mile radius of longitude `88 W` and latitude `30 N`. The example converts distance to radians by dividing distance by the approximate radius of the earth, 3959 miles:

```
db.<collection>.find( { loc : { $geoWithin :
                                 { $centerSphere :
                                    [ [ 88 , 30 ] , 10 / 3959 ]
                      } } } )
```

Proximity to a Point on a Flat Surface Proximity queries return the 100 legacy coordinate pairs closest to the defined point and sort the results by distance. Use either the `$near` (page 801) operator or `geoNear` (page 850) command. Both require a `2d` index.

The `$near` (page 801) operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                         { $near : [ <x> , <y> ]
                      } } )
```

For examples, see `$near` (page 801).

The `geoNear` (page 850) command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The `geoNear` (page 850) command offers more options and returns more information than does the `$near` (page 801) operator. To run the command, see `geoNear` (page 850).

Exact Matches on a Flat Surface You can use the `db.collection.find()` (page 951) method to query for an exact match on a location. These queries use the following syntax:

```
db.<collection>.find( { <location field>: [ <x> , <y> ] } )
```

This query will return any documents with the value of `[ <x> , <y> ]`.

`2dsphere` Indexes

New in version 2.4.

A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both *GeoJSON* objects and as legacy coordinate pairs. The index supports legacy coordinate pairs by converting the data to the GeoJSON `Point` type.

The `2dsphere` index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity.

A *compound* (page 311) `2dsphere` index can reference multiple location and non-location fields within a collection's documents. You can arrange the fields in any order.

The default datum for an earth-like sphere in MongoDB 2.4 is *WGS84*. Coordinate-axis order is **longitude, latitude**.

Important: MongoDB allows *only one* geospatial index per collection. You can create either a `2dsphere` **or** a *2d* (page 341) per collection.

Important: You cannot use a `2dsphere` index as a shard key when sharding a collection. However, you can create and maintain a geospatial index on a sharded collection by using a different field as the shard key.

Store GeoJSON Objects

New in version 2.4.

MongoDB supports the following GeoJSON objects:

- *Point*
- *LineString*
- *Polygon*

In order to index GeoJSON data, you must store the data in a location field that you name. The location field contains a subdocument with a `type` field specifying the GeoJSON object type and a `coordinates` field specifying the object's coordinates. Always store coordinates `longitude, latitude` order.

Use the following syntax:

```
{ <location field> : { type : "<GeoJSON type>" ,
                       coordinates : <coordinates>
} }
```

The following example stores a GeoJSON `Point`:

```
{ loc : { type : "Point" ,
          coordinates : [ 40, 5 ]
} }
```

The following example stores a GeoJSON `LineString`:

```
{ loc : { type : "LineString" ,
          coordinates : [ [ 40 , 5 ] , [ 41 , 6 ] ]
} }
```

Polygons consist of an array of GeoJSON `LinearRing` coordinate arrays. These `LinearRings` are closed `LineStrings`. Closed `LineStrings` have at least four coordinate pairs and specify the same position as the first and last coordinates.

The following example stores a GeoJSON `Polygon` with an exterior ring and no interior rings (or holes). Note the first and last coordinate pair with the `[ 0 , 0 ]` coordinate:

```
{ loc :
   { type : "Polygon" ,
     coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ]
} }
```

For Polygons with multiple rings:

- The first described ring must be the exterior ring.
- The exterior ring cannot self-intersect.
- Any interior ring must be entirely contained by the outer ring.
- Interior rings cannot intersect or overlap each other. Interior rings can share an edge.

The following document represents a polygon with an interior ring as GeoJSON:

```
{ loc :
   { type : "Polygon" ,
     coordinates : [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
                     [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ] ]
} }
```

Create a `2dsphere` Index

To create a geospatial index for GeoJSON-formatted data, use the `ensureIndex()` (page 949) method and set the value of the location field for your collection to `2dsphere`. A `2dsphere` index can be a *compound index* (page 311) and does not require the location field to be the first field indexed.

To create the index use the following syntax:

```
db.points.ensureIndex( { <location field> : "2dsphere" } )
```

The following are four example commands for creating a `2dsphere` index:

```
db.points.ensureIndex( { loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , type : 1 } )
db.points.ensureIndex( { rating : 1 , loc : "2dsphere" } )
db.points.ensureIndex( { loc : "2dsphere" , rating : 1 , category : -1 } )
```

The first example creates a simple geospatial index on the location field `loc`. The second example creates a compound index where the second field contains non-location data. The third example creates an index where the location field is not the primary field: the location field does not have to be the first field in a `2dsphere` index. The fourth example creates a compound index with three fields. You can include as many fields as you like in a `2dsphere` index.

Query a `2dsphere` Index

The following sections describe queries supported by the `2dsphere` index. For an overview of recommended geospatial queries, see *Geospatial Query Compatibility* (page 807).

GeoJSON Objects Bounded by a Polygon The `$geoWithin` (page 800) operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                           { $geometry :
                             { type : "Polygon" ,
                               coordinates : [ <coordinates> ]
                       } } } } )
```

The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates : [ [
                                          [ 0 , 0 ] ,
                                          [ 3 , 6 ] ,
                                          [ 6 , 1 ] ,
                                          [ 0 , 0 ]
                                        ] ]
                } } } } )
```

Intersections of GeoJSON Objects New in version 2.4.

The `$geoIntersects` (page 801) operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty. This includes documents that have a shared edge.

The `$geoIntersects` (page 801) operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoIntersects :
                           { $geometry :
                             { type : "<GeoJSON object type>" ,
                               coordinates : [ <coordinates> ]
                       } } } } )
```

The following example uses `$geoIntersects` (page 801) to select all indexed points and shapes that intersect with the polygon defined by the `coordinates` array.

```
db.places.find( { loc :
                  { $geoIntersects :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates: [ [
                                         [ 0 , 0 ] ,
                                         [ 3 , 6 ] ,
                                         [ 6 , 1 ] ,
```

```
                                    [ 0 , 0 ]
                                  ] ]
                } } } } )
```

Proximity to a GeoJSON Point Proximity queries return the 100 points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a `2dsphere` index.

To query for proximity to a GeoJSON point, use either the `$near` (page 801) operator or `geoNear` (page 850) command. Distance is in meters.

The `$near` (page 801) uses the following syntax:

```
db.<collection>.find( { <location field> :
                         { $near :
                           { $geometry :
                              { type : "Point" ,
                                coordinates : [ <longitude> , <latitude> ] } ,
                             $maxDistance : <distance in meters>
                      } } } )
```

For examples, see `$near` (page 801).

The `geoNear` (page 850) command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The `geoNear` (page 850) command offers more options and returns more information than does the `$near` (page 801) operator. To run the command, see `geoNear` (page 850).

Points within a Circle Defined on a Sphere To select all grid coordinates in a "spherical cap" on a sphere, use `$geoWithin` (page 800) with the `$centerSphere` (page 805) operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distances in a 2d Index Using Spherical Geometry* (page 349).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                           { $centerSphere :
                              [ [ <x>, <y> ] , <radius> ] }
                      } } )
```

The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude `88 W` and latitude `30 N`. The example converts the distance, 10 miles, to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $centerSphere :
                       [ [ 88 , 30 ] , 10 / 3959 ]
                } } } )
```

Haystack Indexes

A haystack index is a special index that is optimized to return results over small areas. Haystack indexes improve performance on queries that use flat geometry.

For queries that use spherical geometry, a **2dsphere index is a better option** than a haystack index. 2dsphere indexes allow field reordering; haystack indexes require the first field to be the location field. Also, haystack indexes are only usable via commands and so always return all results at once.

Haystack indexes create "buckets" of documents from the same geographic area in order to improve performance for queries limited to that area. Each bucket in a haystack index contains all the documents within a specified proximity to a given longitude and latitude.

This document describes how to:

- *Create a Haystack Index* (page 348)
- *Query a Haystack Index* (page 349)

Create a Haystack Index

To build a haystack index, use the `bucketSize` option when creating the index. A `bucketSize` of `5` creates an index that groups location values that are within 5 units of the specified longitude and latitude. The `bucketSize` also determines the granularity of the index. You can tune the parameter to the distribution of your data so that in general you search only very small regions. The areas defined by buckets can overlap. A document can exist in multiple buckets.

A haystack index can reference two fields: the location field and a second field. The second field is used for exact matches. Haystack indexes return documents based on location and an exact match on a single additional criterion. These indexes are not necessarily suited to returning the closest documents to a particular location.

To build a haystack index, use the following syntax:

```
db.coll.ensureIndex( { <location field> : "geoHaystack" ,
                       <additional field> : 1 } ,
                     { bucketSize : <bucket value> } )
```

Example

If you have a collection with documents that contain fields similar to the following:

```
{ _id : 100, pos: { lng : 126.9, lat : 35.2 } , type : "restaurant"}
{ _id : 200, pos: { lng : 127.5, lat : 36.1 } , type : "restaurant"}
{ _id : 300, pos: { lng : 128.0, lat : 36.7 } , type : "national park"}
```

The following operations create a haystack index with buckets that store keys within 1 unit of longitude or latitude.

```
db.places.ensureIndex( { pos : "geoHaystack", type : 1 } ,
                       { bucketSize : 1 } )
```

This index stores the document with an `_id` field that has the value `200` in two different buckets:

- In a bucket that includes the document where the `_id` field has a value of `100`
- In a bucket that includes the document where the `_id` field has a value of `300`

To query using a haystack index you use the `geoSearch` (page 851) command. See *Query a Haystack Index* (page 349).

By default, queries that use a haystack index return 50 documents.

Query a Haystack Index A haystack index is a special `2d` geospatial index that is optimized to return results over small areas. To create a haystack index see *Create a Haystack Index* (page 348).

To query a haystack index, use the `geoSearch` (page 851) command. You must specify both the coordinates and the additional field to `geoSearch` (page 851). For example, to return all documents with the value `restaurant` in the `type` field near the example point, the command would resemble:

```
db.runCommand( { geoSearch : "places" ,
                 search : { type: "restaurant" } ,
                 near : [-74, 40.74] ,
                 maxDistance : 10 } )
```

Note: Haystack indexes are not suited to queries for the complete list of documents closest to a particular location. The closest documents could be more distant compared to the bucket size.

Note: *Spherical query operations* (page 349) are not currently supported by haystack indexes.

The `find()` (page 951) method and `geoNear` (page 850) command cannot access the haystack index.

Calculate Distances in a `2d` Index Using Spherical Geometry

Note: While basic queries using spherical distance are supported by the `2d` index, consider moving to a `2dsphere` index if your data is primarily longitude and latitude.

The `2d` index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

- `$nearSphere` (page 802)
- `$centerSphere` (page 805)
- `$near` (page 801)
- `geoNear` (page 850) command with the `{ spherical: true }` option.

Important: These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- *distance to radians*: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- *radians to distance*: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The radius of the Earth is approximately `3,959` miles or `6,371` kilometers.

The following query would return documents from the `places` collection within the circle described by the center `[ -74, 40.74 ]` with a radius of `100` miles:

```
db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ] ,
                                                        100 / 3959 ] } } } )
```

You may also use the `distanceMultiplier` option to the `geoNear` (page 850) to convert radians in the `mongod` (page 1049) process, rather than in your application code. See *distance multiplier* (page 350).

The following spherical query, returns all documents in the collection `places` within `100` miles from the point `[ -74, 40.74 ]`.

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true
               }  )
```

The output of the above command would be:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 0.01853688938212826,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

Warning: Spherical queries that wrap around the poles or at the transition from `-180` to `180` longitude raise an error.

Note: While the default Earth-like bounds for geospatial indexes are between `-180` inclusive, and `180`, valid values for latitude are between `-90` and `90`.

Distance Multiplier

The `distanceMultiplier` option of the `geoNear` (page 850) command returns distances only after multiplying the results by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Using `distanceMultiplier` in spherical queries provides results from the `geoNear` (page 850) command that do not need radian-to-distance conversion. The following example uses `distanceMultiplier` in the `geoNear` (page 850) command with a *spherical* (page 349) example:

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true,
                 distanceMultiplier: 3959
               }  )
```

The output of the above operation would resemble the following:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 73.46525170413567,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

`2d` Index Internals

This document provides a more in-depth explanation of the internals of MongoDB's `2d` geospatial indexes. This material is not necessary for normal operations or application development but may be useful for troubleshooting and for further understanding.

Calculation of Geohash Values for `2d` Indexes

When you create a geospatial index on *legacy coordinate pairs*, MongoDB computes *geohash* values for the coordinate pairs within the specified *location range* (page 342) and then indexes the geohash values.

To calculate a geohash value, recursively divide a two-dimensional map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

```
01  11

00  10
```

These two-bit values (`00`, `01`, `10`, and `11`) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of `00`. The top left quadrant would have the geohash of `01`. The bottom right and top right would have a geohash of `10` and `11`, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is `11`, and the geohash for the sub-quadrants would be (clockwise from the top left): `1101`, `1111`, `1110`, and `1100`, respectively.

To calculate a more precise geohash, continue dividing the sub-quadrant and concatenate the two-bit identifier for each division. The more "bits" in the hash identifier for a given point, the smaller possible area that the hash can describe and the higher the resolution of the geospatial index.

Multi-location Documents for `2d` Indexes

New in version 2.0: Support for multiple locations in a document.

While `2d` geospatial indexes do not support more than one set of coordinates in a document, you can use a *multi-key index* (page 312) to index multiple coordinate pairs in a single document. In the simplest example you may have a field (e.g. `locs`) that holds an array of coordinates, as in the following example:

```
{ _id : ObjectId(...),
  locs : [ [ 55.5 , 42.3 ] ,
           [ -74 , 44.74 ] ,
           { lng : 55.5 , lat : 42.3 } ]
}
```

The values of the array may be either arrays, as in `[ 55.5, 42.3 ]`, or embedded documents, as in `{ lng : 55.5 , lat : 42.3 }`.

You could then create a geospatial index on the `locs` field, as in the following:

```
db.places.ensureIndex( { "locs": "2d" } )
```

You may also model the location data as a field inside of a sub-document. In this case, the document would contain a field (e.g. `addresses`) that holds an array of documents where each document has a field (e.g. `loc:`) that holds location coordinates. For example:

```
{ _id : ObjectId(...),
  name : "...",
  addresses : [ {
                 context : "home" ,
                 loc : [ 55.5, 42.3 ]
                } ,
                {
                 context : "home",
                 loc : [ -74 , 44.74 ]
                }
              ]
}
```

You could then create the geospatial index on the `addresses.loc` field as in the following example:

```
db.records.ensureIndex( { "addresses.loc": "2d" } )
```

For documents with multiple coordinate values, queries may return the same document multiple times if more than one indexed coordinate pair satisfies the query constraints. Use the `uniqueDocs` parameter to `geoNear` (page 850) or the `$uniqueDocs` (page 807) operator with `$geoWithin` (page 800).

To include the location field with the distance field in multi-location document queries, specify `includeLocs: true` in the `geoNear` (page 850) command.

See also:

Geospatial Query Compatibility (page 807)

CHAPTER 26

Text Indexing

New in version 2.4: `text` indexes were added in 2.4 as a beta feature.

26.1 Text Search

New in version 2.4.

26.1.1 Overview

Text search supports the search of string content in documents of a collection. Text search introduces a new *text* (page 318) index type and a new `text` (page 856) command.

The text search process:

- tokenizes and stems the search term(s) during both the index creation and the text command execution.
- assigns a score to each document that contains the search term in the indexed fields. The score determines the relevance of a document to a given search query.

By default, `text` (page 856) command returns at most the top 100 matching documents as determined by the scores.

Important: Before you can create a text index or *run the text command* (page 360), you need to manually enable the text search. See *Enable Text Search* (page 353) for information on how to enable the text search feature.

Enable Text Search

New in version 2.4.

The *text search* (page 353) is currently a *beta* feature. As a beta feature:

- You need to explicitly enable the feature before *creating a text index* (page 354) or using the `text` (page 856) command.
- To enable text search on *replica sets* (page 373) and *sharded clusters* (page 495), you need to enable on **each and every** `mongod` (page 1049) for replica sets and on **each and every** `mongos` (page 1061) for sharded clusters.

Warning:

- Do **not** enable or use text search on production systems.
- Text indexes have significant storage requirements and performance costs. See *text Indexes* (page 318) for more information.

You can enable the text search feature at startup with the `textSearchEnabled` (page 1132) parameter:

```
mongod --setParameter textSearchEnabled=true
```

You may prefer to set the `textSearchEnabled` (page 1132) parameter in the *configuration file* (page 1115).

Additionally, you can enable the feature in the `mongo` (page 1066) shell with the `setParameter` (page 894) command. This command does **not** propagate from the primary to the secondaries. You must enable on **each and every** `mongod` (page 1049) for replica sets.

Note: You must set the parameter every time you start the server. You may prefer to add the parameter to the *configuration files* (page 1115).

26.1.2 Storage Requirements and Performance Costs

`text` indexes have the following storage requirements and performance costs:

- `text` indexes change the space allocation method for all future record allocations in a collection to `usePowerOf2Sizes` (page 892).
- `text` indexes can be large. They contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a `text` index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large `text` index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors. See the *recommended settings* (page 193).
- `text` indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, `text` indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.

26.1.3 Create a `text` Index

To perform text search, create a `text` index on the field or fields whose value is a string or an array of string elements. To create a `text` index, use the `db.collection.ensureIndex()` (page 949) method with a document that contains field and value pairs where the value is the string literal `text`.

Important:

- Text indexes have significant storage requirements and performance costs. See *Storage Requirements and Performance Costs* (page 354) for more information.
- A collection can have at most **one** `text` index.

The following tutorials offer examples on `text` index creation patterns:

Create `text` Index on Multiple Fields

You can create a `text` index on the field or fields whose value is a string or an array of string elements. When creating a `text` index on multiple fields, you can specify the individual fields or you can wildcard specifier (`$**`).

Index Specific Fields

The following example creates a `text` index on the fields `subject` and `content`:

```
db.collection.ensureIndex(
                           {
                             subject: "text",
                             content: "text"
                           }
                         )
```

This `text` index catalogs all string data in the `subject` field and the `content` field, where the field value is either a string or an array of string elements.

Index All Fields

To allow for text search on all fields with string content, use the wildcard specifier (`$**`) to index all fields that contain string content.

The following example indexes any string value in the data of every field of every document in `collection` and names the index `TextIndex`:

```
db.collection.ensureIndex(
                           { "$**": "text" },
                           { name: "TextIndex" }
                         )
```

Specify a Language for Text Index

The default language associated with the indexed data determines the list of stop words and the rules for the stemmer and tokenizer. The default language for the indexed data is `english`.

To specify a different language, use the `default_language` option when creating the `text` index. See *Text Search Languages* (page 860) for the languages available for `default_language`.

The following example creates a `text` index on the `content` field and sets the `default_language` to `spanish`:

```
db.collection.ensureIndex(
                           { content : "text" },
                           { default_language: "spanish" }
                         )
```

See also:

Create a text Index on a Multi-language Collection (page 356)

Specify `text` Index Name to Avoid Name Length Limit

The default name for the index consists of each indexed field name concatenated with `_text`. For example, the following command creates a `text` index on the fields `content`, `users.comments`, and `users.profiles`:

```
db.collection.ensureIndex(
                           {
                             content: "text",
                             "users.comments": "text",
                             "users.profiles": "text"
```

```
                                  }
                                )
```

The default name for the index is:

```
"content_text_users.comments_text_users.profiles_text"
```

To avoid creating an index with a name that exceeds the `index name length limit` (page 1140), you can pass the `name` option to the `db.collection.ensureIndex()` (page 949) method:

```
db.collection.ensureIndex(
                                  {
                                    content: "text",
                                    "users.comments": "text",
                                    "users.profiles": "text"
                                  },
                                  {
                                    name: "MyTextIndex"
                                  }
                                )
```

Note: To drop the `text` index, use the index name. To get the name of an index, use `db.collection.getIndexes()` (page 956).

Create a `text` Index on a Multi-language Collection

Specify the Index Language within the Document

If a collection contains documents that are in different languages, include a field in the documents that contain the language to use:

- If you include a field named `language` in the document, by default, the `ensureIndex()` (page 949) method will use the value of this field to override the default language.
- To use a field with a name other than `language`, you must specify the name of this field to the `ensureIndex()` (page 949) method with the `language_override` option.

See *Text Search Languages* (page 860) for a list of supported languages.

Include the `language` Field

Include a field `language` that specifies the language to use for the individual documents.

For example, the documents of a multi-language collection `quotes` contain the field `language`:

```
{ _id: 1, language: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, language: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, language: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote`:

```
db.quotes.ensureIndex( { quote: "text" } )
```

- For the documents that contain the `language` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.

- For documents that do not contain the `language` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` (page 856) command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Use any Field to Specify the Language for a Document

Include a field that specifies the language to use for the individual documents. To use a field with a name other than `language`, include the `language_override` option when creating the index.

For example, the documents of a multi-language collection `quotes` contain the field `idioma`:

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, idioma: "spanish", quote: "Nada hay más surreal que la realidad." }
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }
```

Create a `text` index on the field `quote` with the `language_override` option:

```
db.quotes.ensureIndex( { quote : "text" },
                       { language_override: "idioma" } )
```

- For the documents that contain the `idioma` field, the `text` index uses that language to determine the stop words and the rules for the stemmer and the tokenizer.
- For documents that do not contain the `idioma` field, the index uses the default language, which is English, to determine the stop words and rules for the stemmer and the tokenizer.

For example, the Spanish word `que` is a stop word. So the following `text` (page 856) command would not match any document:

```
db.quotes.runCommand( "text", { search: "que", language: "spanish" } )
```

Control Results of Text Search with Weights

By default, the `text` (page 856) command returns matching documents based on scores, from highest to lowest. For a `text` index, the *weight* of an indexed field denotes the significance of the field relative to the other indexed fields in terms of the score. The score for a given word in a document is derived from the weighted sum of the frequency for each of the indexed fields in that document.

The default weight is 1 for the indexed fields. To adjust the weights for the indexed fields, include the `weights` option in the `db.collection.ensureIndex()` (page 949) method.

> **Warning:** Choose the weights carefully in order to prevent the need to reindex.

A collection `blog` has the following documents:

```
{ _id: 1,
  content: "This morning I had a cup of coffee.",
  about: "beverage",
  keywords: [ "coffee" ]
}

{ _id: 2,
  content: "Who doesn't like cake?",
```

```
  about: "food",
  keywords: [ "cake", "food", "dessert" ]
}
```

To create a `text` index with different field weights for the `content` field and the `keywords` field, include the `weights` option to the `ensureIndex()` (page 949) method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```
db.blog.ensureIndex(
                     {
                       content: "text",
                       keywords: "text",
                       about: "text"
                     },
                     {
                       weights: {
                                  content: 10,
                                  keywords: 5,
                                },
                       name: "TextIndex"
                     }
                   )
```

The `text` index has the following fields and weights:

- `content` has a weight of 10,
- `keywords` has a weight of 5, and
- `about` has the default weight of 1.

These weights denote the relative significance of the indexed fields to each other. For instance, a term match in the `content` field has:

- `2` times (i.e. `10:5`) the impact as a term match in the `keywords` field and
- `10` times (i.e. `10:1`) the impact as a term match in the `about` field.

Limit the Number of Index Entries Scanned for Text Search

The `text` (page 856) command includes the `filter` option to further restrict the results of a text search. For a `filter` that specifies equality conditions, this tutorial demonstrates how to perform text searches on only those documents that match the `filter` conditions, as opposed to performing a text search first on all the documents and then matching on the `filter` condition.

Consider a collection `inventory` that contains the following documents:

```
{ _id: 1, dept: "tech", description: "a fun green computer" }
{ _id: 2, dept: "tech", description: "a wireless red mouse" }
{ _id: 3, dept: "kitchen", description: "a green placemat" }
{ _id: 4, dept: "kitchen", description: "a red peeler" }
{ _id: 5, dept: "food", description: "a green apple" }
{ _id: 6, dept: "food", description: "a red potato" }
```

A common use case is to perform text searches by individual departments, such as:

```
db.inventory.runCommand( "text", {
                                   search: "green",
                                   filter: { dept : "kitchen" }
```

```
                                    }
                       )
```

To limit the text search to scan only those documents within a specific `dept`, create a compound index that specifies an ascending/descending index key on the field `dept` and a `text` index key on the field `description`:

```
db.inventory.ensureIndex(
                          {
                            dept: 1,
                            description: "text"
                          }
                        )
```

Important:

- The ascending/descending index keys must be listed before, or prefix, the `text` index keys.
- By prefixing the `text` index fields with ascending/descending index fields, MongoDB will **only** index documents that have the prefix fields.
- You cannot include *multi-key* (page 312) index fields or *geospatial* (page 318) index fields.
- The `text` (page 856) command **must** include the `filter` option that specifies an **equality** condition for the prefix fields.

Then, the text search within a particular department will limit the scan of indexed documents. For example, the following `text` (page 856) command scans only those documents with `dept` equal to `kitchen`:

```
db.inventory.runCommand( "text", {
                                   search: "green",
                                   filter: { dept : "kitchen" }
                                 }
                       )
```

The returned result includes the statistics that shows that the command scanned 1 document, as indicated by the `nscanned` field:

```
{

  "queryDebugString" : "green||||||",
  "language" : "english",
  "results" : [
                {
                  "score" : 0.75,
                  "obj" : {
                            "_id" : 3,
                            "dept" : "kitchen",
                            "description" : "a green placemat"
                          }
                }
              ],
  "stats" : {
              "nscanned" : 1,
              "nscannedObjects" : 0,
              "n" : 1,
              "nfound" : 1,
              "timeMicros" : 211
            },
```

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

For more information on the result set, see *Output* (page 859).

Return Text Queries Using Only a `text` Index

To create a `text` index that can *cover queries* (page 322):

1. Append scalar index fields to a `text` index, as in the following example which specifies an ascending index key on `username`:

   ```
   db.collection.ensureIndex( { comments: "text",
                                username: 1 } )
   ```

   Warning: You cannot include *multi-key* (page 312) index field or *geospatial* (page 318) index field.

2. Use the `project` option in the `text` (page 856) to return only the fields in the index, as in the following:

   ```
   db.quotes.runCommand( "text", { search: "tomorrow",
                                   project: { username: 1,
                                              _id: 0
                                            }
                                 }
                       )
   ```

Note: By default, the `_id` field is included in the result set. Since the example index did not include the `_id` field, you must explicitly exclude the field in the `project` document.

26.1.4 `text` Command

The `text` (page 856) command can search for words and phrases. The command matches on the complete stemmed words. For example, if a document field contains the word `blueberry`, a search on the term `blue` will not match the document. However, a search on either `blueberry` or `blueberries` will match.

For information and examples on various text search patterns, see *Search String Content for Text* (page 360).

Search String Content for Text

In 2.4, you can enable the text search feature to create `text` indexes and issue text queries using the `text` (page 856).

The following tutorial offers various query patterns for using the text search feature.

The examples in this tutorial use a collection `quotes` that has a `text` index on the fields `quote` that contains a string and `related_quotes` that contains an array of string elements.

Search for a Term

The following command searches for the word `TOMORROW`:

```
db.quotes.runCommand( "text", { search: "TOMORROW" } )
```

Because `text` (page 856) command is case-insensitive, the text search will match the following document in the `quotes` collection:

```
{
  "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),
  "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",
  "related_quotes" : [
                       "is this a dagger which I see before me",
                       "the handle toward my hand?"
                     ],
  "src" : {
            "title" : "Macbeth",
            "from" : "Act V, Scene V"
          },
  "speaker" : "macbeth"
}
```

Match Any of the Search Terms

If the search string is a space-delimited text, `text` (page 856) command performs a logical `OR` search on each term and returns documents that contains any of the terms.

For example, the search string `"tomorrow largo"` searches for the term `tomorrow` **OR** the term `largo`:

```
db.quotes.runCommand( "text", { search: "tomorrow largo" } )
```

The command will match the following documents in the `quotes` collection:

```
{
  "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),
  "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",
  "related_quotes" : [
                       "is this a dagger which I see before me",
                       "the handle toward my hand?"
                     ],
  "src" : {
            "title" : "Macbeth",
            "from" : "Act V, Scene V"
          },
  "speaker" : "macbeth"
 }

 {
   "_id" : ObjectId("50ecf0cd8abea0fda30ceab4"),
   "quote" : "Es tan corto el amor y es tan largo el olvido.",
   "related_quotes" : [
                        "Como para acercarla mi mirada la busca.",
                        "Mi corazón la busca, y ella no está conmigo."
                      ],
   "speaker" : "Pablo Neruda",
   "src" : {
             "title" : "Veinte poemas de amor y una canción desesperada",
             "from" : "Poema 20"
           }
  }
```

Match Phrases

To match the exact phrase that includes a space(s) as a single term, escape the quotes.

For example, the following command searches for the exact phrase "and tomorrow":

```
db.quotes.runCommand( "text", { search: "\"and tomorrow\"" } )
```

If the search string contains both phrase and individual terms, the text (page 856) command performs a compound logical AND of the phrases with the compound logical OR of the single terms.

For example, the following command contains a search string that contains the individual terms corto and largo as well as the phrase \"and tomorrow\":

```
db.quotes.runCommand( "text", { search: "corto largo \"and tomorrow\"" } )
```

The text (page 856) command performs the equivalent to the following logical operation:

```
(corto OR largo OR tomorrow) AND ("and tomorrow")
```

Match Some Words But Not Others

A *negated* term is a term that is prefixed by a minus sign -. If you negate a term, the text (page 856) command will exclude the documents that contain those terms from the results.

Note: If the search text contains *only* negated terms, the text (page 856) command will not return any results.

The following example returns those documents that contain the term tomorrow but **not** the term petty.

```
db.quotes.runCommand( "text" , { search: "tomorrow -petty" } )
```

Limit the Number of Matching Documents in the Result Set

Note: The result from the text (page 856) command must fit within the maximum BSON Document Size (page 1139).

By default, the text (page 856) command will return up to 100 matching documents, from highest to lowest scores. To override this default limit, use the limit option in the text (page 856) command, as in the following example:

```
db.quotes.runCommand( "text", { search: "tomorrow", limit: 2 } )
```

The text (page 856) command will return at most 2 of the *highest scoring* results.

The limit can be any number as long as the result set fits within the maximum BSON Document Size (page 1139).

Specify Which Fields to Return in the Result Set

In the text (page 856) command, use the project option to specify the fields to include (1) or exclude (0) in the matching documents.

Note: The _id field is always returned unless explicitly excluded in the project document.

The following example returns only the _id field and the src field in the matching documents:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                project: { "src": 1 } } )
```

Search with Additional Query Conditions

The `text` (page 856) command can also use the `filter` option to specify additional query conditions.

The following example will return the documents that contain the term `tomorrow` **AND** the `speaker` is `macbeth`:

```
db.quotes.runCommand( "text", { search: "tomorrow",
                                filter: { speaker : "macbeth" } } )
```

See also:

Limit the Number of Index Entries Scanned for Text Search (page 358)

Search for Text in Specific Languages

You can specify the language that determines the tokenization, stemming, and removal of stop words, as in the following example:

```
db.quotes.runCommand( "text", { search: "amor", language: "spanish" } )
```

See also:

Create a text Index on a Multi-language Collection (page 356)

See *Text Search Languages* (page 860) for a list of supported languages.

Text Search Output

The `text` (page 856) command returns a document that contains the result set.

See *Output* (page 859) for information on the output.

26.1.5 Text Search Output

The `text` (page 856) command returns a document that contains the result set.

See *Output* (page 859) for information on the output.

Part VIII

Replication

A *replica set* in MongoDB is a group of `mongod` (page 1049) processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

CHAPTER 27

Replication Introduction

Replication is the process of synchronizing data across multiple servers.

27.1 Purpose of Replication

Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

In some cases, you can use replication to increase read capacity. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

27.2 Replication in MongoDB

A replica set is a group of `mongod` (page 1049) instances that host the same data set. One `mongod` (page 1049), the primary, receives all write operations. All other instances, secondaries, apply operations from the primary so that they have the same data set.

The **primary** accepts all write operations from clients. Replica set can have only one primary. Because only one member can accept write operations, replica sets provide **strict consistency**. To support replication, the primary logs all changes to its data sets in its *oplog* (page 405). See *primary* (page 374) for more information.

The **secondaries** replicate the primary's oplog and apply the operations to their data sets. Secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary. By default, clients read from the primary, however, clients can specify a *read preferences* (page 398) to send read operations to secondaries. See *secondaries* (page 374) for more information

You may add an extra `mongod` (page 1049) instance a replica set as an **arbiter**. Arbiters do not maintain a data set. Arbiters only exist to vote in elections. If your replica set has an even number of members, add an arbiter to obtain a majority of votes in an election for primary. Arbiters do not require dedicated hardware. See *arbiter* (page 381) for more information.

Note: An **arbiter** will always be an arbiter. A **primary** may step down and become a **secondary**. A **secondary** may become the primary during an election.

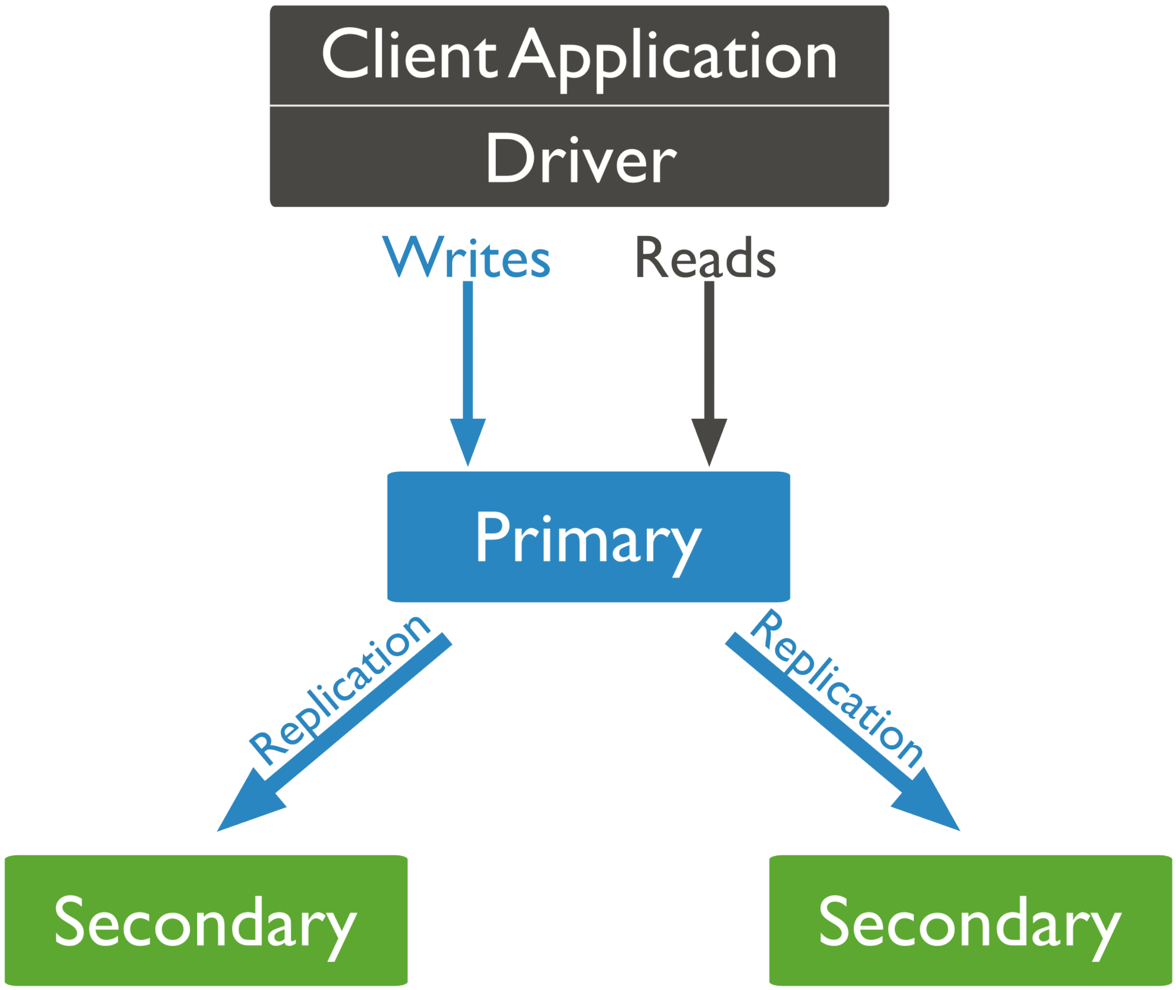


Figure 27.1: Diagram of default routing of reads and writes to the primary.

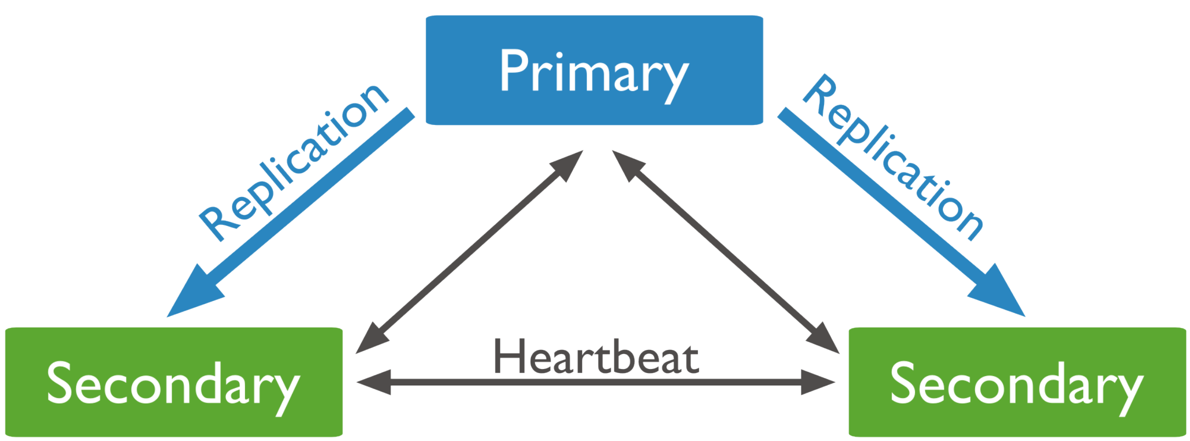


Figure 27.2: Diagram of a 3 member replica set that consists of a primary and two secondaries.

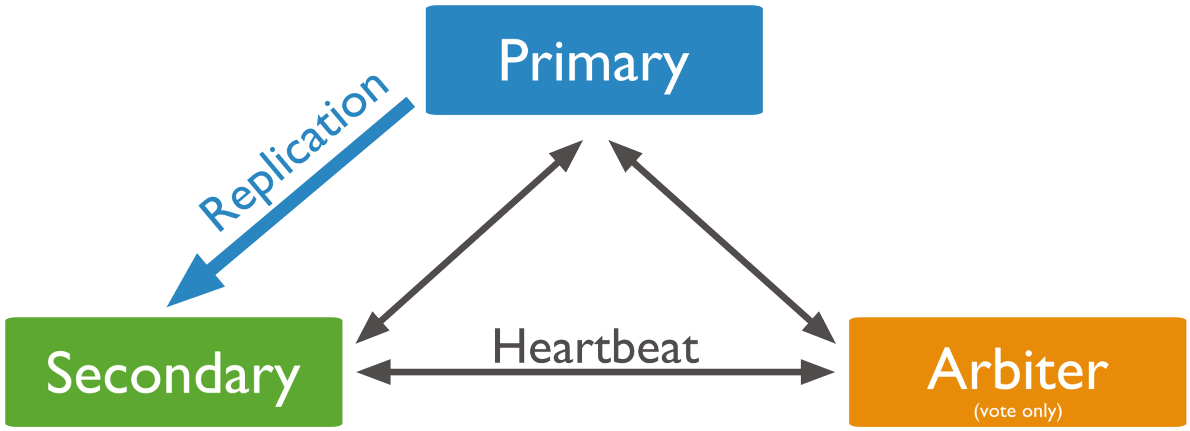


Figure 27.3: Diagram of 3 member replica set that consists of a primary, a secondary, and an arbiter.

27.2.1 Asynchronous Replication

Secondaries apply operations from the primary asynchronously. By applying operations after the primary, sets can continue to function without some members. However, as a result secondaries may not return the most current data to clients.

See *Replica Set Oplog* (page 405) and *Replica Set Data Synchronization* (page 406) for more information. See *Read Preference* (page 398) for more on read operations and secondaries.

27.2.2 Automatic Failover

When a primary does not communicate with the other members of the set for more than 10 seconds, the replica set will attempt to select another member to become the new primary. The first secondary that receives a majority of the votes becomes primary.

See *Replica Set Elections* (page 389) and *Rollbacks During Replica Set Failover* (page 393) for more information.

27.2.3 Additional Features

Replica sets provide a number of options to support application needs. For example, you may deploy a replica set with *members in multiple data centers* (page 388), or control the outcome of elections by adjusting the `priority` (page 475) of some members. Replica sets also support dedicated members for reporting, disaster recovery, or backup functions.

See *Priority 0 Replica Set Members* (page 378), *Hidden Replica Set Members* (page 379) and *Delayed Replica Set Members* (page 379) for more information.

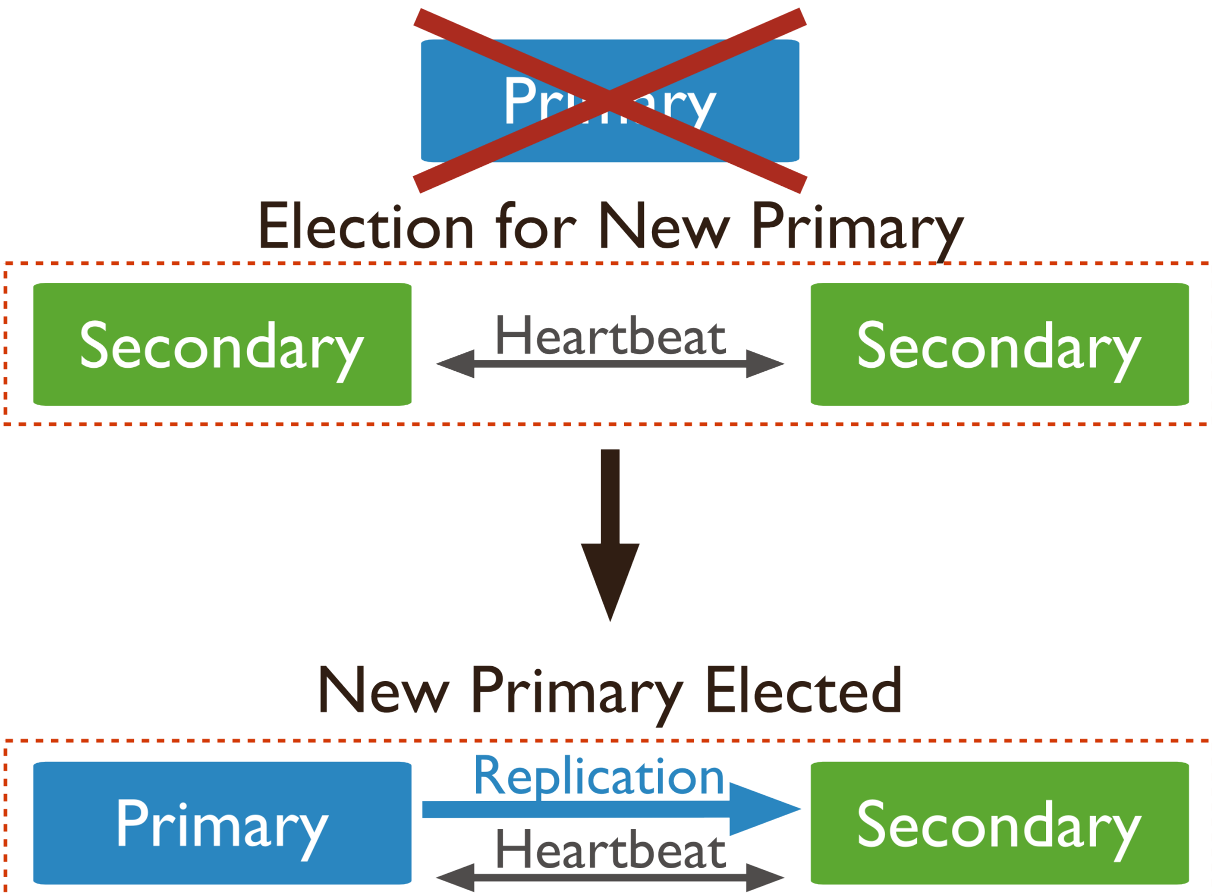


Figure 27.4: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

CHAPTER 28

Replication Concepts

These documents describe and provide examples of replica set operation, configuration, and behavior. For an overview of replication, see *Replication Introduction* (page 369). For documentation of the administration of replica sets, see *Replica Set Tutorials* (page 415). The *Replication Reference* (page 461) documents commands and operations specific to replica sets.

Replica Set Members **(page 374)** Introduces the components of replica sets.

- ***Replica Set Primary*** **(page 374)** The primary is the only member of a replica set that accepts write operations.
- ***Replica Set Secondary Members*** **(page 374)** Secondary members replicate the primary's data set and accept read operations. If the set has no primary, a secondary can become primary.
- ***Priority 0 Replica Set Members*** **(page 378)** Priority 0 members are secondaries that cannot become the primary.
- ***Hidden Replica Set Members*** **(page 379)** Hidden members are secondaries that are invisible to applications. These members support dedicated workloads, such as reporting or backup.
- ***Replica Set Arbiter*** **(page 381)** An arbiter does not maintain a copy of the data set but participate in elections.

Replica Set Deployment Architectures **(page 382)** Introduces architectural considerations related to replica sets deployment planning.

- ***Three Member Replica Sets*** **(page 384)** Three-member replica sets provide the minimum recommended architecture for a replica set.
- ***Replica Sets with Four or More Members*** **(page 386)** Four or more member replica sets provide greater redundancy and can support greater distribution of read operations and dedicated functionality.

Replica Set High Availability **(page 388)** Presents the details of the automatic failover and recovery process with replica sets.

- ***Replica Set Elections*** **(page 389)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.
- ***Read Preference*** **(page 398)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

Replication Processes **(page 405)** Mechanics of the replication process and related topics.

Master Slave Replication **(page 408)** Master-slave replication provided redundancy in early versions of MongoDB. Replica sets replace master-slave for most use cases.

28.1 Replica Set Members

A *replica set* in MongoDB is a group of `mongod` (page 1049) processes that provide redundancy and high availability. The members of a replica set are:

***Primary* (page ??).** The *primary* receives all write operations.

***Secondaries* (page ??).** Secondaries replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be *non-voting* (page 392) or *priority 0* (page 378).

You can also maintain an *arbiter* (page **??**) as part of a replica set. Arbiters do not keep a copy of the data. However, arbiters play a role in the elections that select a primary if the current primary is unavailable.

A replica set can have up to 12 members. [1] However, only 7 members can vote at a time.

The minimum requirements for a replica set are: A *primary* (page **??**), a *secondary* (page **??**), and an *arbiter* (page **??**). Most deployments, however, will keep three members that store data: A *primary* (page **??**) and two *secondary members* (page **??**).

28.1.1 Replica Set Primary

The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the *primary* and then records the operations on the primary's *oplog* (page 405). *Secondary* (page **??**) members replicate this log and apply the operations to their data sets.

In the following three-member replica set, the primary accepts all write operations. Then the secondaries replicate the oplog to apply to their data sets.

All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. See *Read Preference* (page 398) for details on changing the default read behavior.

The replica set can have at most one primary. If the current primary becomes unavailable, an election determines the new primary. See *Replica Set Elections* (page 389) for more details.

In the following 3-member replica set, the primary becomes unavailable. This triggers an election which selects one of the remaining secondaries as the new primary.

28.1.2 Replica Set Secondary Members

A secondary maintains a copy of the *primary's* data set. To replicate data, a secondary applies operations from the primary's *oplog* (page 405) to its own data set in an asynchronous process. A replica set can have one or more secondaries.

The following three-member replica set has two secondary members. The secondaries replicate the primary's oplog and apply the operations to their data sets.

Although clients cannot write data to secondaries, clients can read data from secondary members. See *Read Preference* (page 398) for more information on how clients direct read operations to replica sets.

A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an *election* to choose with of the secondaries becomes the new primary.

In the following three-member replica set, the primary becomes unavailable. This triggers an election where one of the remaining secondaries becomes the new primary.

[1] While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use *master-slave* (page 408) replication. Master-slave replication lacks the automatic failover capabilities.

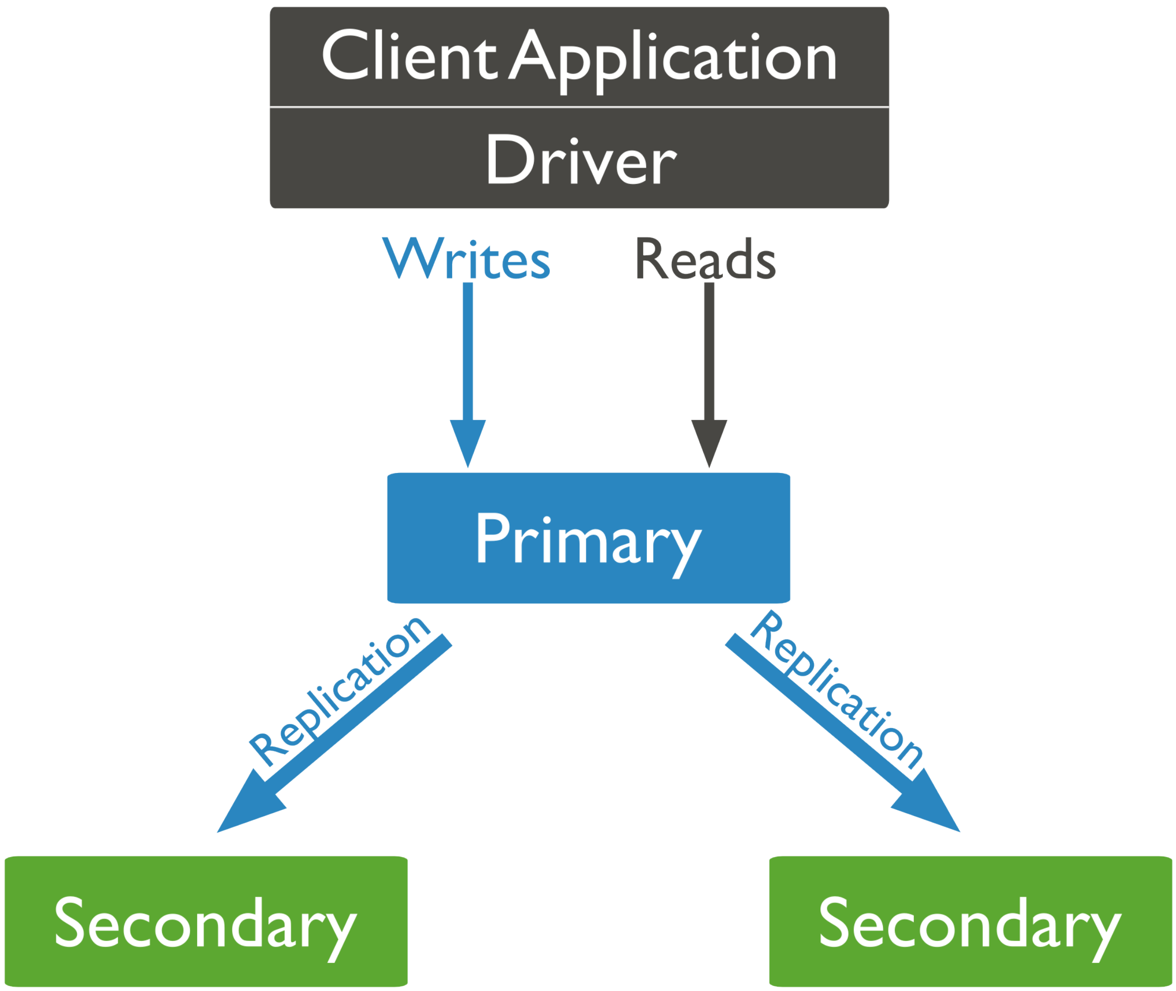


Figure 28.1: Diagram of default routing of reads and writes to the primary.

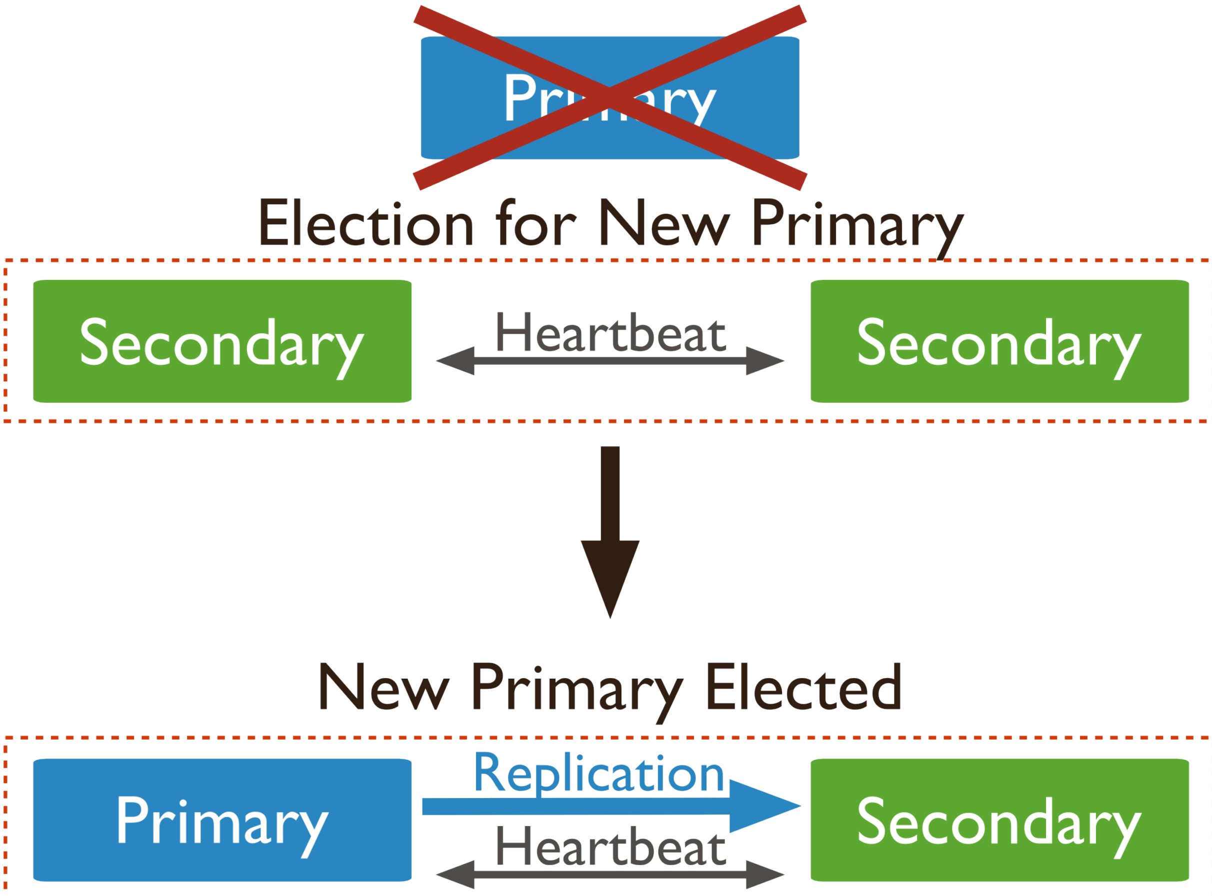


Figure 28.2: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

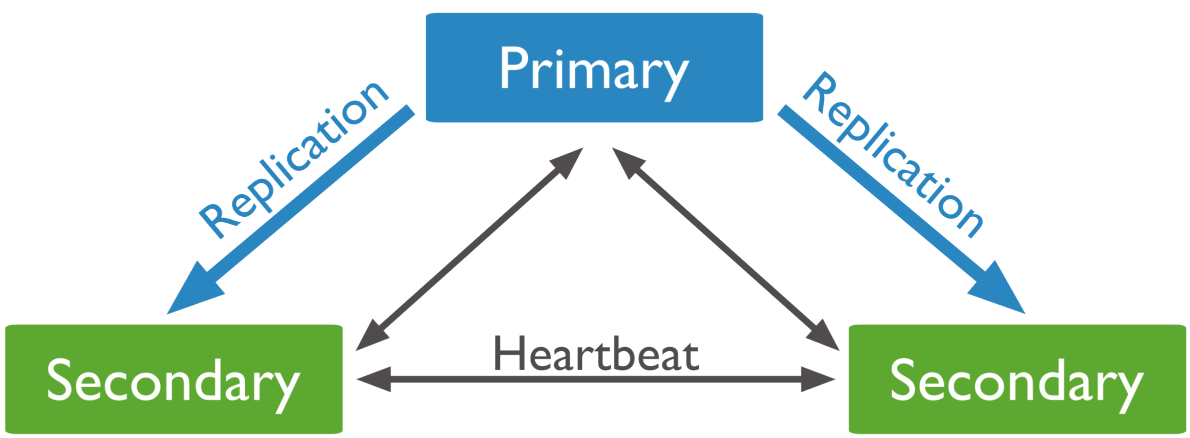


Figure 28.3: Diagram of a 3 member replica set that consists of a primary and two secondaries.

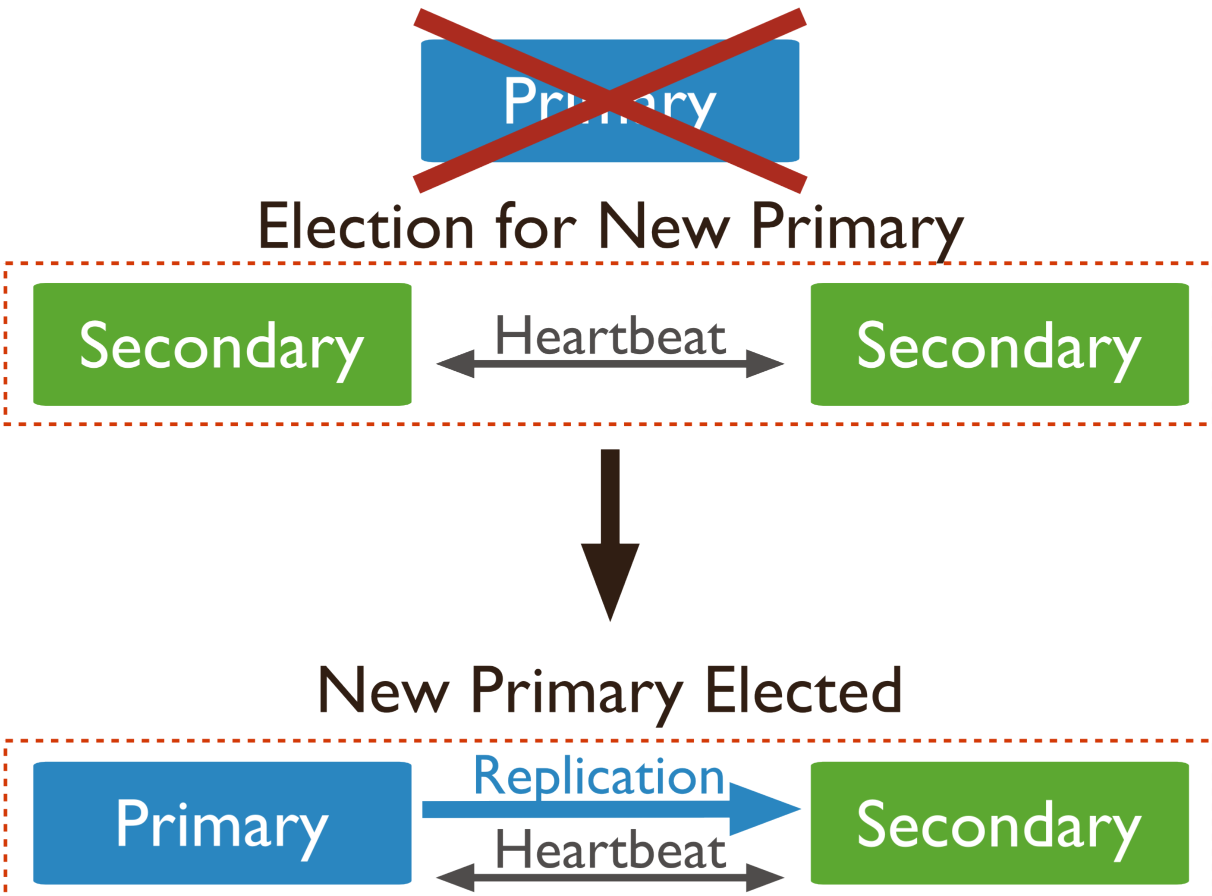


Figure 28.4: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

See *Replica Set Elections* (page 389) for more details.

You can configure a secondary member for a specific purpose. You can configure a secondary to:

- Prevent it from becoming a primary in an election, which allows it to reside in a secondary data center or to serve as a cold standby. See *Priority 0 Replica Set Members* (page 378).
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic. See *Hidden Replica Set Members* (page 379).
- Keep a running "historical" snapshot for use in recovery from certain errors, such as unintentionally deleted databases. See *Delayed Replica Set Members* (page 379).

Priority 0 Replica Set Members

A *priority 0* member is a secondary that **cannot** become *primary*. *Priority 0* members cannot *trigger elections*. Otherwise these members function as normal secondaries. A *priority 0* member maintains a copy of the data set, accepts read operations, and votes in elections. Configure a *priority 0* member to prevent *secondaries* from becoming primary, which is particularly useful in multi-data center deployments.

In a three-member replica set, in one data center hosts the primary and a secondary. A second data center hosts one *priority 0* member that cannot become primary.

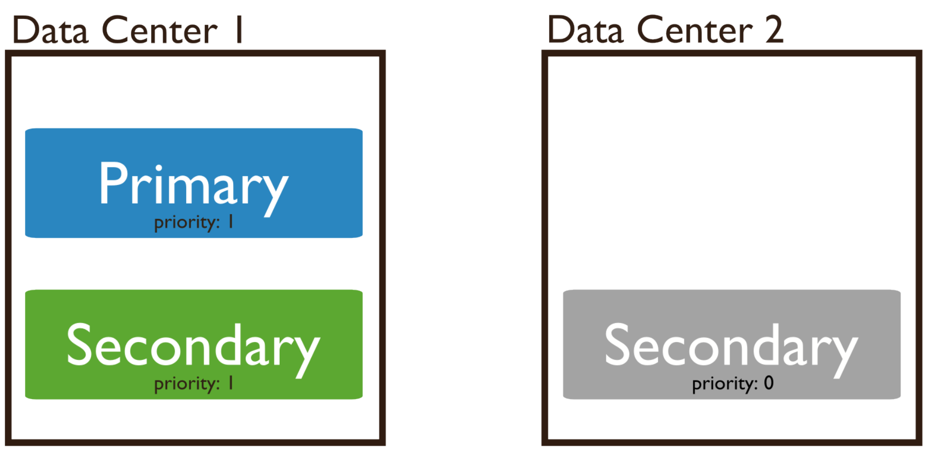


Figure 28.5: Diagram of a 3 member replica set distributed across two data centers. Replica set includes a priority 0 member.

Priority 0 Members as Standbys

A *priority 0* member can function as a standby. In some replica sets, it might not be possible to add a new member in a reasonable amount of time. A standby member keeps a current copy of the data to be able to replace an unavailable member.

In many cases, you need not set standby to *priority 0*. However, in sets with varied hardware or *geographic distribution* (page 388), a *priority 0* standby ensures that only qualified members become primary.

A *priority 0* standby may also be valuable for some members of a set with different hardware or workload profiles. In these cases, deploy a member with *priority 0* so it can't become primary. Also consider using an *hidden member* (page 379) for this purpose.

If your set already has seven voting members, also configure the member as *non-voting* (page 392).

Priority 0 Members and Failover

When configuring a *priority 0* member, consider potential failover patterns, including all possible network partitions. Always ensure that your main data center contains both a quorum of voting members and contains members that are eligible to be primary.

Configuration

To configure a *priority 0* member, see *Prevent Secondary from Becoming Primary* (page 432).

Hidden Replica Set Members

A hidden member maintains a copy of the *primary's* data set but is **invisible** to client applications. Hidden members are ideal for workloads with different usage patterns from the other members in the *replica set*. Hidden members are also *priority 0 members* (page 378) and **cannot become primary**. The `db.isMaster()` (page 1008) method does not display hidden members. Hidden members, however, **do vote** in *elections* (page 389).

In the following five-member replica set, all four secondary members have copies of the primary's data set, but one of secondary members is hidden.

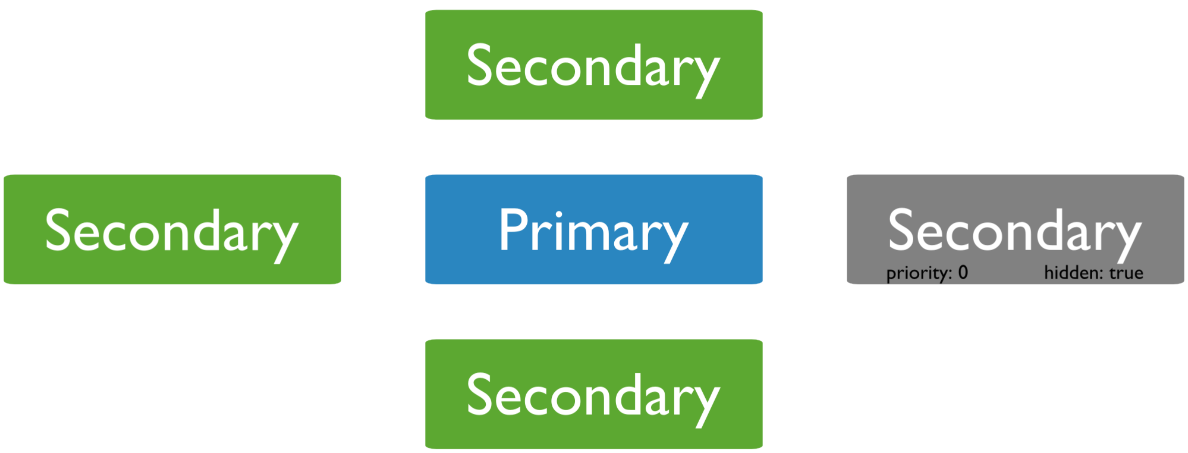


Figure 28.6: Diagram of a 5 member replica set with a hidden priority 0 member.

Secondary reads do not reach a hidden member, so the member receives no traffic beyond what replication requires. Use hidden members for instances that require separation from normal traffic. For example, you might use hidden members for instances dedicated to reporting or to backups.

For dedicated backup, ensure that the hidden member has low network latency to network to the primary or likely primary. Ensure that the *replication lag* is minimal or non-existent. For more information about backing up MongoDB databases, see *Backup Strategies for MongoDB Systems* (page 133).

To configure a hidden member, see *Configure a Hidden Replica Set Member* (page 433).

Delayed Replica Set Members

Delayed members contain copies of a *replica set's* data set. However, a delayed member's data set reflects an earlier, or delayed, state of the set. For example, if the current time is 09:52 and a member has a delay of an hour, the delayed member has no operation more recent than 08:52.

Because delayed members are a "rolling backup" or a running "historical" snapshot of the data set, they may help you recover from various kinds of human error. For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.

Requirements

Delayed members:

- **Must be** *priority 0* (page 378) members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** *hidden* (page 379) members. Always prevent applications from seeing and querying delayed members.
- *do* vote in *elections* for primary.

Delayed members apply operations from the *oplog* on a delay. When choosing the amount of delay, consider that the amount of delay:

- must be is equal to or greater than your maintenance windows.
- must be *smaller* than the capacity of the oplog. For more information on oplog size, see *Oplog Size* (page 405).

Example

In the following 5-member replica set, the primary and all secondaries have copies of the data set. One member applies operations with a delay of 3600 seconds, or an hour. This delayed member is also *hidden* and is a *priority 0 member*.

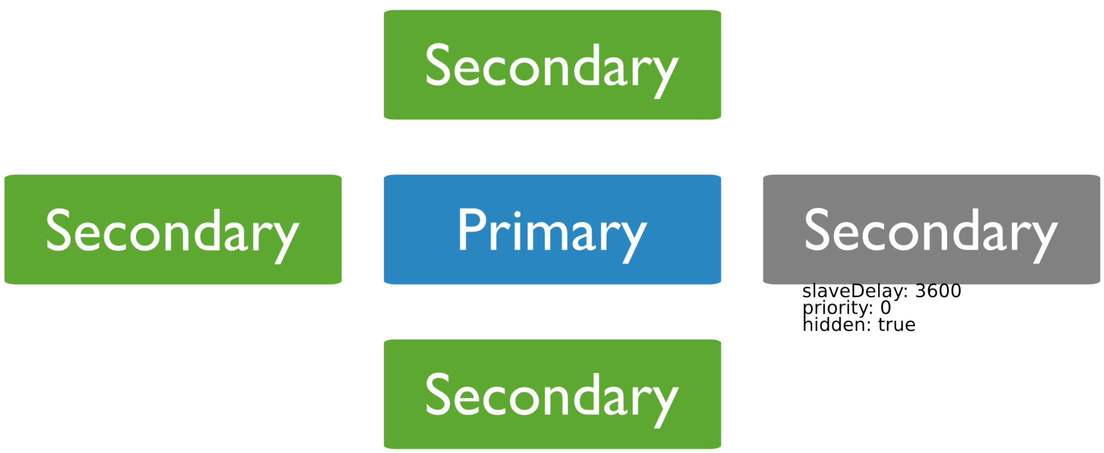


Figure 28.7: Diagram of a 5 member replica set with a hidden delayed priority 0 member.

Configuration

A delayed member has its `priority` (page 475) equal to `0`, `hidden` (page 475) equal to `true`, and its `slaveDelay` (page 476) equal to the number of seconds of delay:

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
```

```
    "priority" : 0,
    "slaveDelay" : <seconds>,
    "hidden" : true
}
```

To configure a delayed member, see *Configure a Delayed Replica Set Member* (page 434).

28.1.3 Replica Set Arbiter

An arbiter does **not** have a copy of data set and **cannot** become a primary. Replica sets may have arbiters to add a vote in *elections of for primary* (page 389). Arbiters allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.

Important: Do not run an arbiter on systems that also host the primary or the secondary members of the replica set.

Only add an arbiter to sets with even numbers of members. If you add an arbiter to a set with an odd number of members, the set may suffer from tied *elections*. To add an arbiter, see *Add an Arbiter to Replica Set* (page 425).

Example

For example, in the following replica set, an arbiter allows the set to have an odd number of votes for elections:

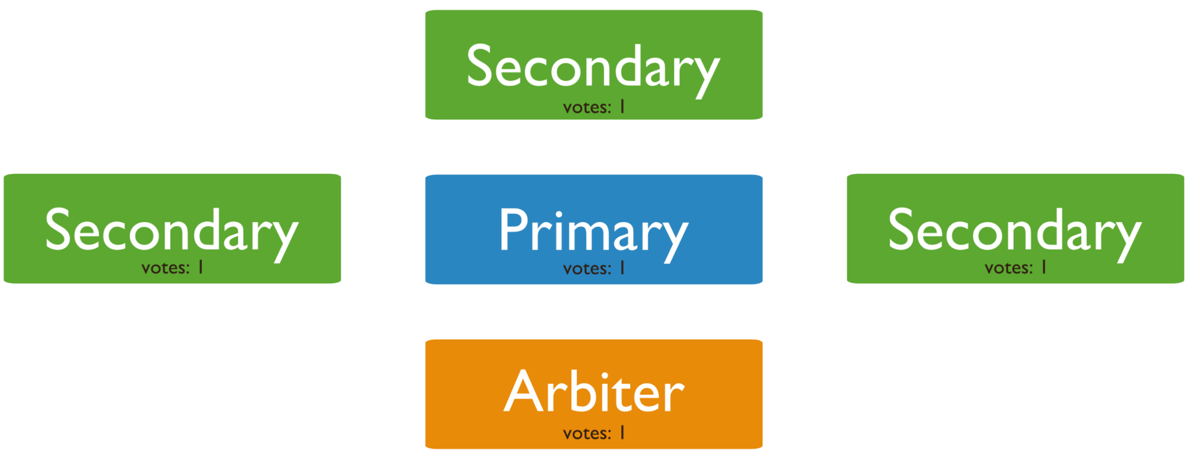


Figure 28.8: Diagram of a four member replica set plus an arbiter for odd number of votes.

Security

Authentication

When running with `auth` (page 1118), arbiters exchange credentials with other members of the set to authenticate. MongoDB encrypts the authentication process. The MongoDB authentication exchange is cryptographically secure.

Arbiters, use `keyfiles` to authenticate to the replica set.

Communication

The only communication between arbiters and other set members are: votes during elections, heartbeats, and configuration data. These exchanges are not encrypted.

However, if your MongoDB deployment uses SSL, MongoDB will encrypt *all* communication between replica set members. See *Connect to MongoDB with SSL* (page 179) for more information.

As with all MongoDB components, run arbiters on in trusted network environments.

28.2 Replica Set Deployment Architectures

The architecture of a *replica set* affects the set's capacity and capability. This document provides strategies for replica set deployments and describes common architectures.

The standard replica set deployment for production system is a three-member replica set. These sets provide redundancy and fault tolerance. Avoid complexity when possible, but let your application requirements dictate the architecture.

28.2.1 Strategies

Determine the Number of Members

Add members in a replica set according to these strategies.

Deploy an Odd Number of Members

An odd number of members ensures that the replica set is always able to elect a primary. If you have an even number of members, add an arbiter to get an odd number. *Arbiters* do not store a copy of the data and require fewer resources. As a result, you may run an arbiter on an application server or other shared process.

Consider Fault Tolerance

Fault tolerance for a replica set is the number of members that can become unavailable and still leave enough members in the set to elect a primary. In other words, it is the difference between the number of members in the set and the majority needed to elect a primary. Without a primary, a replica set cannot accept write operations. Fault tolerance is an effect of replica set size, but the relationship is not direct. See the following table:

Number of Members.	Majority Required to Elect a New Primary.	Fault Tolerance.
3	2	1
4	3	1
5	3	2
6	4	2

Adding a member to the replica set does not *always* increase the fault tolerance. However, in these cases, additional members can provide support for dedicated functions, such as backups or reporting.

Use Hidden and Delayed Members for Dedicated Functions

Add *hidden* (page 379) or *delayed* (page 379) members to support dedicated functions, such as backup or reporting.

Load Balance on Read-Heavy Deployments

In a deployment with *very* high read traffic, you can improve read throughput by distributing reads to secondary members. As your deployment grows, add or move members to alternate data centers to improve redundancy and availability.

Always ensure that the main facility is able to elect a primary.

Add Capacity Ahead of Demand

The existing members of a replica set must have spare capacity to support adding a new member. Always add new members before the current demand saturates the capacity of the set.

Determine the Distribution of Members

Distribute Members Geographically

To protect your data if your main data center fails, keep at least one member in an alternate data center. Set these members' `priority` (page 475) to 0 to prevent them from becoming primary.

Keep a Majority of Members in One Location

When a replica set has members in multiple data centers, network partitions can prevent communication between data centers. To replicate data, members must be able to communicate to other members.

In an election, members must see each other to create a majority. To ensure that the replica set members can confirm a majority and elect a primary, keep a majority of the set's members in one location.

Target Operations with Tags

Use *replica set tags* (page 444) to ensure that operations replicate to specific data centers. Tags also support targeting read operations to specific machines.

See also:

Data Center Awareness (page 151) and *Operational Segregation in MongoDB Operations and Deployments* (page 151).

Use Journaling to Protect Against Power Failures

Enable journaling to protect data against service interruptions. Without journaling MongoDB cannot recover data after unexpected shutdowns, including power failures and unexpected reboots.

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

28.2.2 Deployment Patterns

The following documents describe common replica set deployment patterns. Other patterns are possible and effective depending on the the application's requirements. If needed, combine features of each architecture in your own deployment:

Three Member Replica Sets (page 384) Three-member replica sets provide the minimum recommended architecture for a replica set.

Replica Sets with Four or More Members (page 386) Four or more member replica sets provide greater redundancy and can support greater distribution of read operations and dedicated functionality.

Geographically Distributed Replica Sets (page 388) Geographically distributed sets include members in multiple locations to protect against facility-specific failures, such as power outages.

Three Member Replica Sets

The minimum architecture of a replica set has three members. A three member replica set can have either three members that hold data, or two members that hold data and an arbiter.

Primary with Two Secondary Members

A replica set with three members that store data has:

- One *primary* (page 374).
- Two *secondary* (page 374) members. Both secondaries can become the primary in an *election* (page 389).

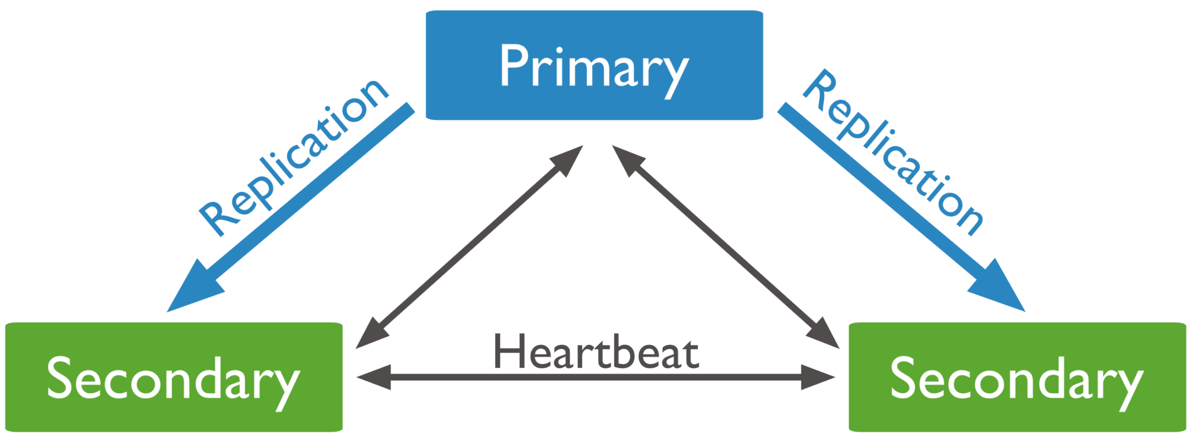


Figure 28.9: Diagram of a 3 member replica set that consists of a primary and two secondaries.

These deployments provide two complete copies of the data set at all times in addition to the primary. These replica sets provide additional fault tolerance and *high availability* (page 388). If the primary is unavailable, the replica set elects a secondary to be primary and continues normal operation. The old primary rejoins the set when available.

Primary with a Secondary and an Arbiter

A three member replica set with a two members that store data has:

- One *primary* (page 374).
- One *secondary* (page 374) member. The secondary can become primary in an *election* (page 389).
- One *arbiter* (page 381). The arbiter only votes in elections.

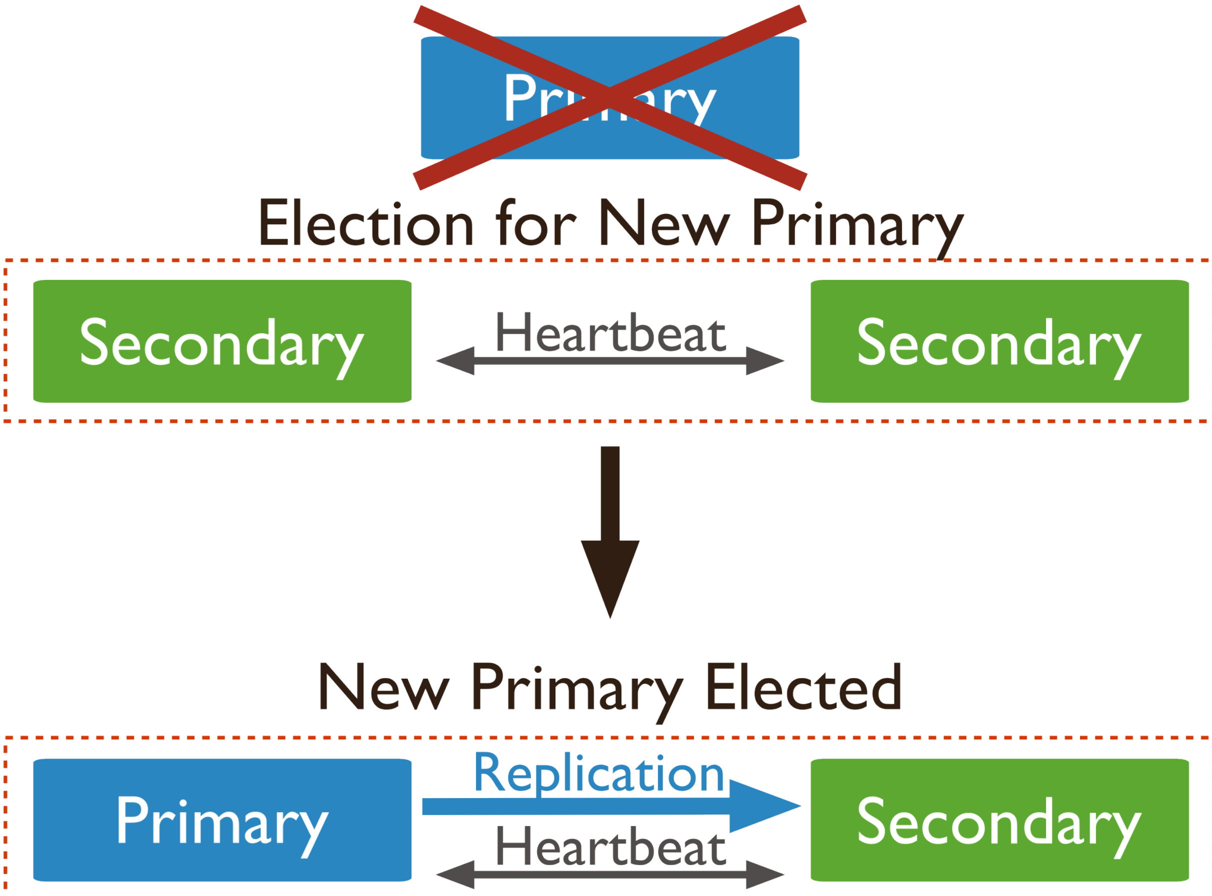


Figure 28.10: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

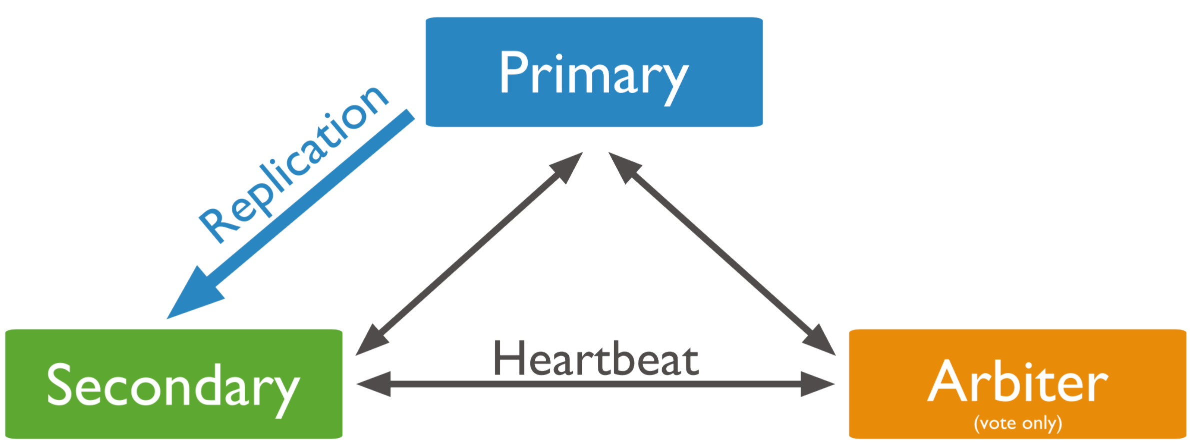


Figure 28.11: Diagram of 3 member replica set that consists of a primary, a secondary, and an arbiter.

Since the arbiter does not hold a copy of the data, these deployments provides only one complete copy of the data. Arbiters require fewer resources, at the expense of more limited redundancy and fault tolerance.

However, a deployment with a primary, secondary, and an arbiter ensures that a replica set remains available if the primary *or* the secondary is unavailable. If the primary is unavailable, the replica set will elect the secondary to be primary.

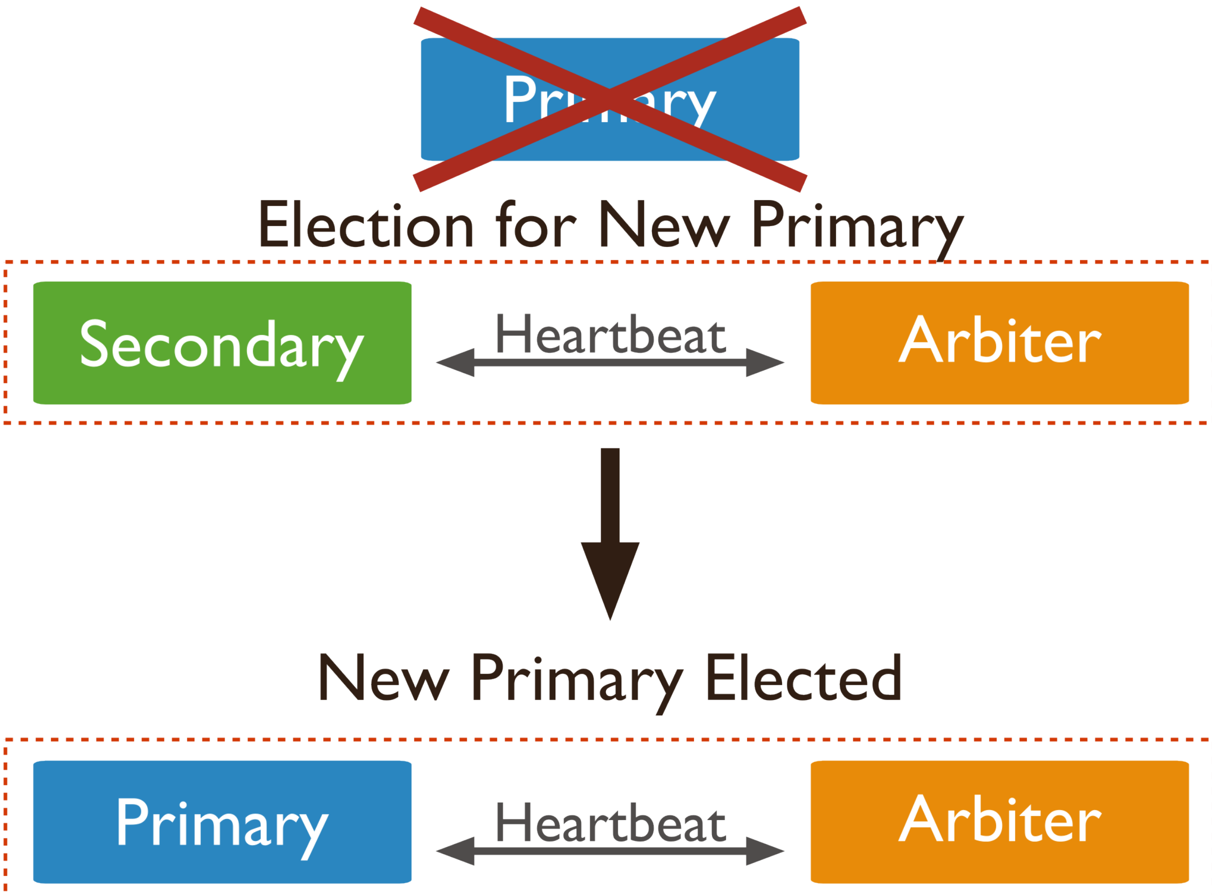


Figure 28.12: Diagram of an election of a new primary. In a three member replica set with a secondary and an arbiter, the primary becomes unreachable. The loss of a primary triggers an election where the secondary becomes new primary.

See also:

Deploy a Replica Set (page 416).

Replica Sets with Four or More Members

Although the standard replica set configuration has three members you can deploy larger sets. Add additional members to a set to increase redundancy or to add capacity for distributing secondary read operations.

When adding members, ensure that:

- The set has an odd number of voting members. If you have an *even* number of voting members, deploy an *arbiter* (page **??**) so that the set has an odd number.

The following replica set needs an arbiter to have an odd number of voting members.

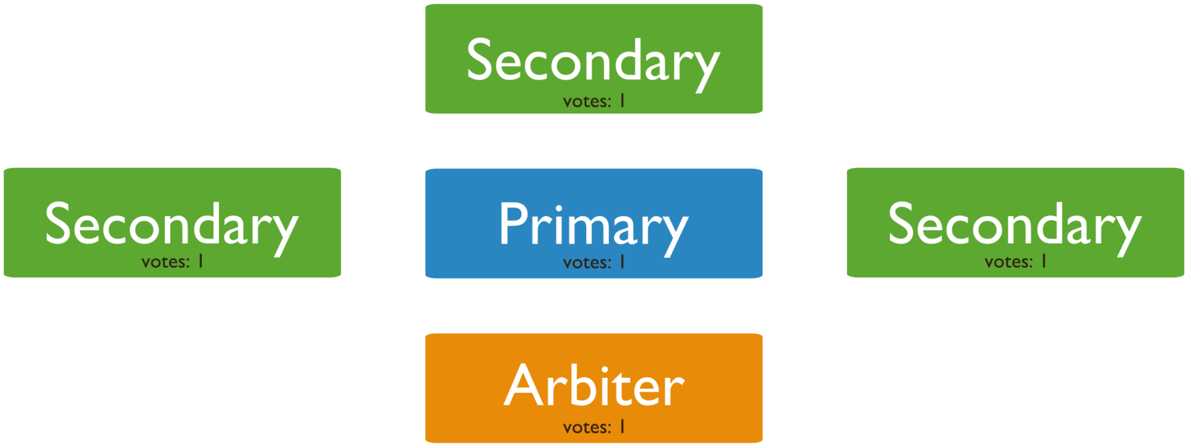


Figure 28.13: Diagram of a four member replica set plus an arbiter for odd number of votes.

- A replica set can have up to 12 members, [2] but only 7 voting members. See *non-voting members* (page 392) for more information.

  The following 9 member replica set has 7 voting members and 2 non-voting members.

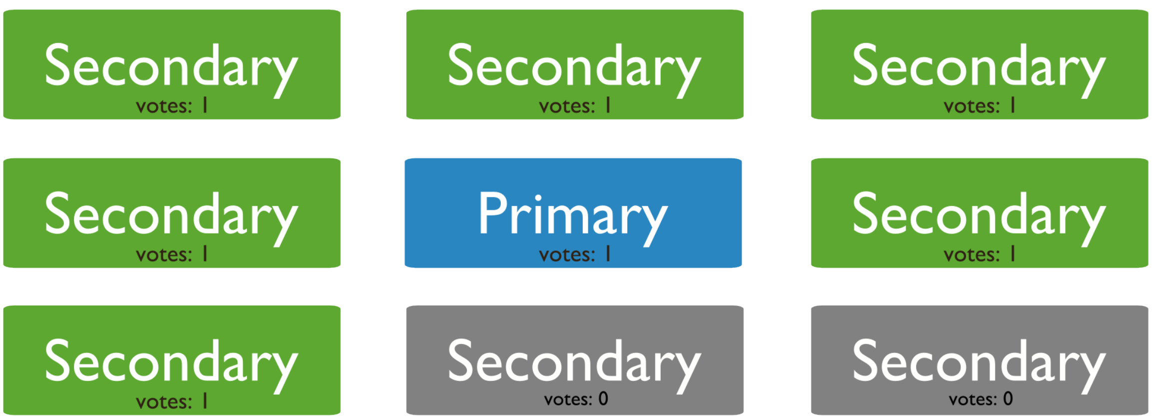


Figure 28.14: Diagram of a 9 member replica set with the maximum of 7 voting members.

- Members that cannot become primary in a *failover* have *priority 0 configuration* (page 378).

  For instance, some members that have limited resources or networking constraints and should never be able to become primary. Configure members that should not become primary to have *priority 0* (page 378). In following replica set, the secondary member in the third data center has a priority of 0:

- A majority of the set's members should be in your applications main data center.

See also:

[2] While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 12 members, you'll need to use *master-slave* (page 408) replication. Master-slave replication lacks the automatic failover capabilities.

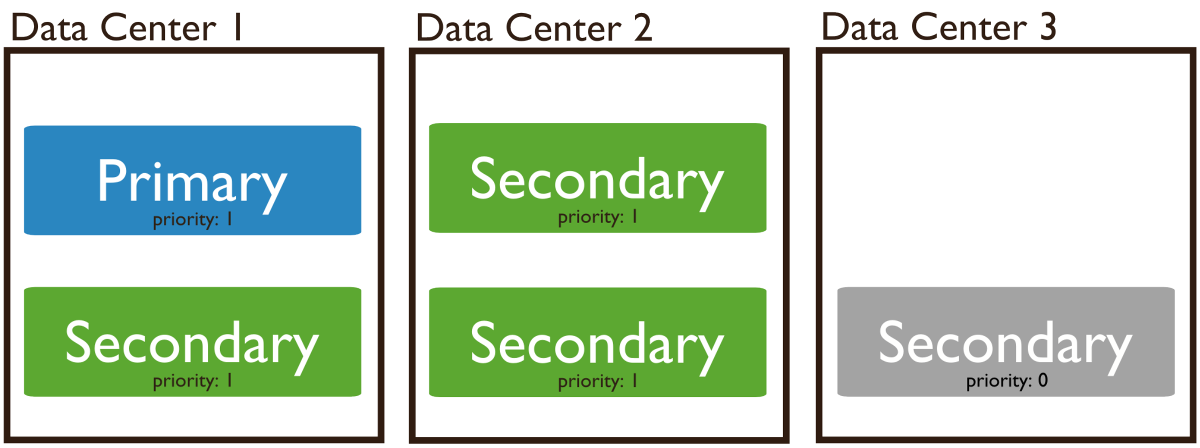


Figure 28.15: Diagram of a 5 member replica set distributed across three data centers. Replica set includes a priority 0 member.

Deploy a Replica Set (page 416), *Add an Arbiter to Replica Set* (page 425), and *Add Members to a Replica Set* (page 427).

Geographically Distributed Replica Sets

Adding members to a replica set in multiple data centers adds redundancy and provides fault tolerance if one data center is unavailable. Members in additional data centers should have a *priority of 0* (page 378) to prevent them from becoming primary.

For example: the architecture of a geographically distributed replica set may be:

- One *primary* in the main data center.
- One *secondary* member in the main data center. This member can become primary at any time.
- One *priority 0* (page 378) member in a second data center. This member cannot become primary.

In the following replica set, the primary and one secondary are in *Data Center 1*, while *Data Center 2* has a *priority 0* (page 378) secondary that cannot become a primary.

If the primary is unavailable, the replica set will elect a new primary from *Data Center 1*. If the data centers cannot connect to each other, the member in *Data Center 2* will not become the primary.

If *Data Center 1* becomes unavailable, you can manually recover the data set from *Data Center 2* with minimal downtime. With sufficient *write concern* (page 395), there will be no data loss.

To facilitate elections, the main data center should hold a majority of members. Also ensure that the set has an odd number of members. If adding a member in another data center results in a set with an even number of members, deploy an *arbiter* (page **??**). For more information on elections, see *Replica Set Elections* (page 389).

See also:

Deploy a Geographically Distributed Replica Set (page 420).

28.3 Replica Set High Availability

Replica sets provide high availability using automatic *failover*. Failover allows a *secondary* members to become

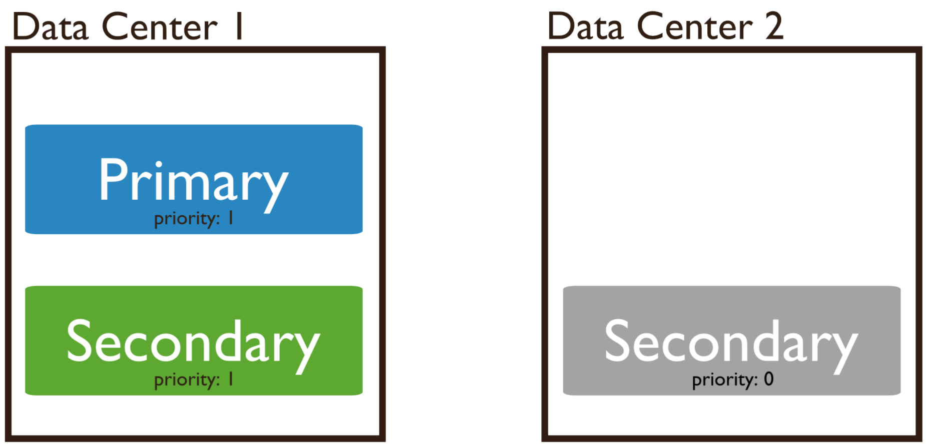


Figure 28.16: Diagram of a 3 member replica set distributed across two data centers. Replica set includes a priority 0 member.

primary if primary is unavailable. Failover, in most situations does not require manual intervention.

Replica set members keep the same data set but are otherwise independent. If the primary becomes unavailable, the replica set holds an *election* (page 389) to select a new primary. In some situations, the failover process may require a *rollback* (page 393). [3]

The deployment of a replica set affects the outcome of failover situations. To support effective failover, ensure that one facility can elect a primary if needed. Choose the facility that hosts the core application systems to host the majority of the replica set. Place a majority of voting members and all the members that can become primary in this facility. Otherwise, network partitions could prevent the set from being able to form a majority.

28.3.1 Failover Processes

The replica set recovers from the loss of a primary by holding an election. Consider the following:

***Replica Set Elections* (page 389)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

***Rollbacks During Replica Set Failover* (page 393)** A rollback reverts write operations on a former primary when the member rejoins the replica set after a failover.

Replica Set Elections

Replica sets use elections to determine which set member will become *primary*. Elections happen after initiating a replica set and anytime the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the *failover process* (page 388).

Important: Elections are essential for independent operation of a replica set; however, elections take time to complete. While an election is in process, the replica set has no primary and cannot accept writes. MongoDB avoids elections unless required.

In the following three-member replica set, the primary is unavailable. The remaining secondaries hold an election to choose a new primary.

[3] Replica sets remove “rollback” data when needed without intervention. Administrators must apply or discard rollback data manually.

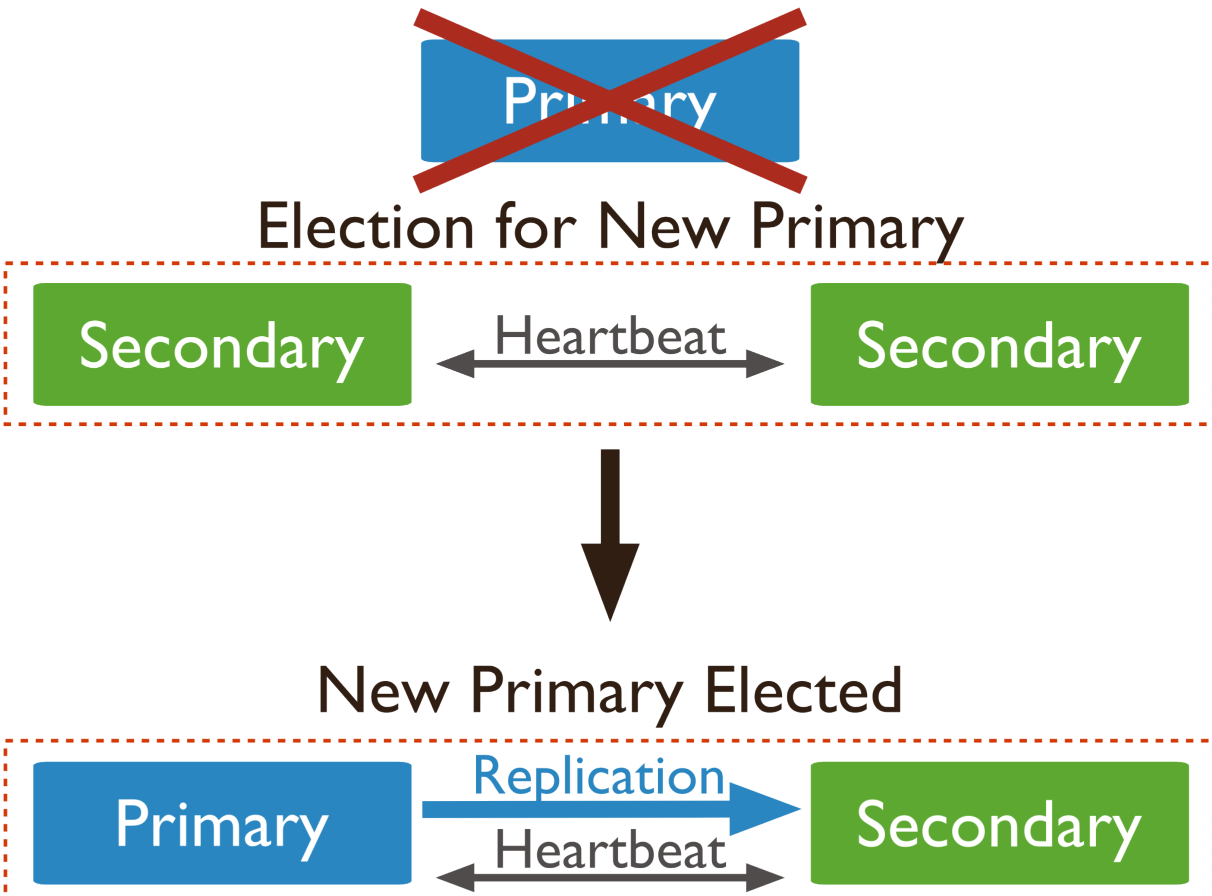


Figure 28.17: Diagram of an election of a new primary. In a three member replica set with two secondaries, the primary becomes unreachable. The loss of a primary triggers an election where one of the secondaries becomes the new primary

Factors and Conditions that Affect Elections

Heartbeats Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.

Priority Comparisons The `priority` (page 475) setting affects elections. Members will prefer to vote for members with the highest priority value.

Members with a priority value of `0` cannot become primary and do not seek election. For details, see *Priority 0 Replica Set Members* (page 378).

A replica set does *not* hold an election as long as the primary has the highest priority value and is within 10 seconds of the latest *oplog* entry. If the member drops to beyond 10 seconds of the latest oplog entry, the set holds an election.

Optime The `optime` (page 866) is the timestamp of the last operation that a member applied from the oplog. A replica set member cannot become primary *unless* it has the highest (i.e. most recent) `optime` (page 866) of any visible member in the set.

Connections A replica set member cannot become primary *unless* it can connect to a majority of the members in the replica set. For the purposes of elections, a majority refers to the total number of *votes*, rather than the total number of members.

If you have a three-member replica set, where every member has one vote, the set can elect a primary as long as two members can connect to each other. If two members are unavailable, the remaining member remains a *secondary* because it cannot connect to a majority of the set's members.

While there is no primary, clients cannot write to the replica set.

Network Partitions Network partitions affect the formation of a majority for an election. If a primary steps down and neither portion of the replica set has a majority the set will **not** elect a new primary. The replica set becomes read-only.

To avoid this situation, place a majority of instances in one data center and a minority of instances in any other data centers combined.

Election Mechanics

Election Triggering Events Replica sets hold an election any time there is no primary. Specifically, following:

- the initiation of a new replica set.
- a secondary loses contact with a primary. Secondaries call for elections when they cannot connect to a primary.
- a primary steps down.

Note: *Priority 0 members* (page 378), do not trigger elections, even when they cannot connect to the primary.

A primary will step down:

- after receiving the `replSetStepDown` (page 868) command.
- if one of the current secondaries is eligible for election *and* has a higher priority.
- if primary cannot contact a majority of the members of the replica set.

Important: When a primary steps down, it closes all open client connections, so that clients don't attempt to write data to a secondary. This helps clients maintain an accurate view of the replica set and helps prevent *rollbacks*.

Participation in Elections Every replica set member has a *priority* that helps determine its eligibility to become a *primary*. In an election, the replica set elects an eligible member with the highest `priority` (page 475) value as primary. By default, all members have a priority of `1` and have an equal chance of becoming primary. In the default, all members also can trigger an election.

You can set the `priority` (page 475) value to weight the election in favor of a particular member or group of members. For example, if you have a *geographically distributed replica set* (page 388), you can adjust priorities so that only members in a specific data center can become primary.

The first member to receive the majority of votes becomes primary. By default, all members have a single vote, unless you modify the `votes` (page 476) setting. *Non-voting members* (page 435) have `votes` (page 476) value of `0`.

The `state` (page 866) of a member also affects its eligibility to vote. Only members in the following states can vote: `PRIMARY`, `SECONDARY`, `RECOVERING`, `ARBITER`, and `ROLLBACK`.

Important: Do not alter the number of votes in a replica set to control the outcome of an election. Instead, modify the `priority` (page 475) value.

Vetoes in Elections All members of a replica set can veto an election, including *non-voting members* (page 392). A member will veto an election:

- If the member seeking an election is not a member of the voter's set.
- If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.
- If the member seeking an election has a lower priority than another member in the set that is also eligible for election.
- If a *priority 0 member* (page 378) [4] is the most current member at the time of the election. In this case, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.
- If the current primary has more recent operations (i.e. a higher `optime` (page 866)) than the member seeking election, from the perspective of the voting member.
- If the current primary has the same or more recent operations (i.e. a higher or equal `optime` (page 866)) than the member seeking election.

Non-Voting Members

Non-voting members hold copies of the replica set's data and can accept read operations from client applications. Non-voting members: do not vote in elections, but **can** *veto* (page 392) an election and become primary.

Because a replica set can have up to 12 members but only up to seven voting members, non-voting members permit a replica set to have more than seven members.

For instance, the following nine-member replica set has seven voting members and two non-voting members.

A non-voting member has a `votes` (page 476) setting equal to `0` in its member configuration:

[4] Remember that *hidden* (page 379) and *delayed* (page 379) imply *priority 0* (page 378) configuration.

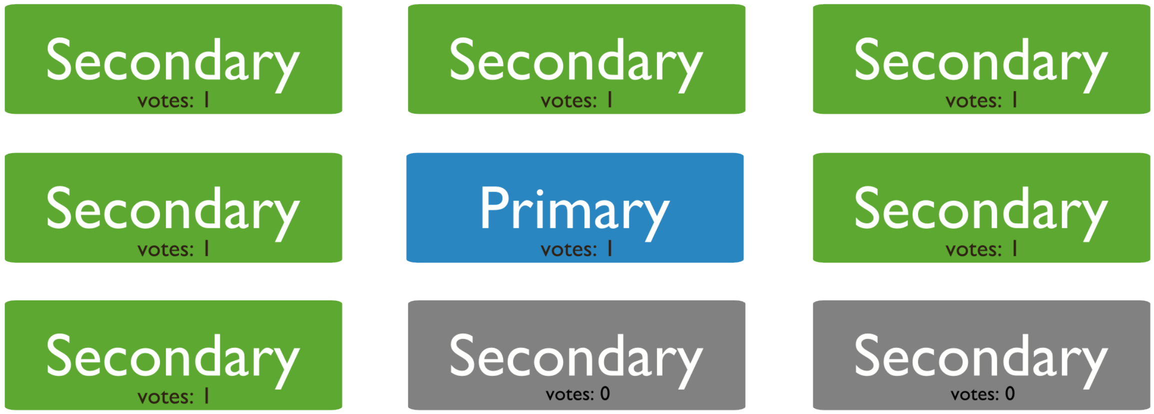


Figure 28.18: Diagram of a 9 member replica set with the maximum of 7 voting members.

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "votes" : 0
}
```

Important: Do **not** alter the number of votes to control which members will become primary. Instead, modify the `priority` (page 475) option. *Only* alter the number of votes in exceptional cases. For example, to permit more than seven secondaries.

When possible, all members should have only one vote. Changing the number of votes can cause ties, deadlocks, and the wrong members from becoming primary.

To configure a non-voting member, see *Configure Non-Voting Replica Set Member* (page 435).

Rollbacks During Replica Set Failover

A rollback reverts write operations on a former *primary* when the member rejoins its *replica set* after a *failover*. A rollback is necessary only if the primary had accepted write operations that the *secondaries* had **not** successfully replicated before the primary stepped down. When the primary rejoins the set as a secondary, it reverts, or "rolls back," its write operations to maintain database consistency with the other members.

MongoDB attempts to avoid rollbacks, which should be rare. When a rollback does occur, it is often the result of a network partition. Secondaries that can not keep up with the throughput of operations on the former primary, increase the size an impact of the rollback.

A rollback does *not* occur if the write operations replicate to another member of the replica set before the primary steps down and if that member remains available and accessible to a majority of the replica set.

Collect Rollback Data

When a rollback does occur, administrators must decide whether to apply or ignore the rollback data. MongoDB writes the rollback data to *BSON* files in the `rollback/` folder under the database's `dbpath` (page 1118) directory. The names of rollback files have the following form:

```
<database>.<collection>.<timestamp>.bson
```

For example:

```
records.accounts.2011-05-09T18-10-04.0.bson
```

Administrators must apply rollback data manually after the member completes the rollback and returns to secondary status. Use *bsondump* (page 1084) to read the contents of the rollback files. Then use `mongorestore` (page 1079) to apply the changes to the new primary.

Avoid Replica Set Rollbacks

To prevent rollbacks, use *replica acknowledged write concern* (page 396) to guarantee that the write operations propagate to the members of a replica set.

Rollback Limitations

A `mongod` (page 1049) instance will not rollback more than 300 megabytes of data. If your system must rollback more than 300 megabytes, you must manually intervene to recover the data. If this is the case, the following line will appear in your `mongod` (page 1049) log:

```
[replica set sync] replSet syncThread: 13410 replSet too much data to roll back
```

In this situation, save the data directly or force the member to perform an initial sync. To force initial sync, sync from a "current" member of the set by deleting the content of the `dbpath` (page 1118) directory for the member that requires a larger rollback.

See also:

Replica Set High Availability (page 388) and *Replica Set Elections* (page 389).

28.4 Replica Set Read and Write Semantics

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent.

By default, in MongoDB, read operations to a replica set return results from the *primary* (page 374) and are *consistent* with the last write operation.

Users may configure *read preference* on a per-connection basis to prefer that the read operations return on the *secondary* members. If clients configure the *read preference* to permit secondary reads, read operations cannot return from *secondary* members that have not replicated more recent updates or operations. When reading from a secondary, a query may return data that reflects a previous state.

This behavior is sometimes characterized as *eventual consistency* because the secondary member's state will *eventually* reflect the primary's state and MongoDB cannot guarantee *strict consistency* for read operations from secondary members.

To guarantee consistency for reads from secondary members, you can configure the *client* and *driver* to ensure that write operations succeed on all members before completing successfully. See *Write Concern* (page 395) for more information. Additionally, such configuration can help prevent *Rollbacks During Replica Set Failover* (page 393) during a failover.

Note: *Sharded clusters* where the shards are also replica sets provide the same operational semantics with regards to write and read operations.

***Write Concern* (page 395)** Write concern is the guarantee an application requires from MongoDB to consider a write operation successful.

***Read Preference* (page 398)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

28.4.1 Write Concern

Write concern is a quality of every write operation issued to a MongoDB deployment, and describes the amount of *concern* the application has for the outcome of the write operation. With weak or disabled write concern, the application can send a write operation to MongoDB and then continue without waiting for a response from the database. With stronger write concerns, write operations wait until MongoDB acknowledges or confirms a successful write operation. MongoDB provides different levels of write concern to better address the specific needs of applications.

See also:

Write Concern Reference (page 57) for a reference of specific write concern configuration. Also consider *Write Operations* (page 53) for a general overview of write operations with MongoDB.

Note: After the *driver write concern change* (page 1223) all officially supported MongoDB drivers enable write concern by default.

Types of Write Concern

Clients issue write operations with some level of *write concern*, which describes the level of concern or guarantee the server will provide in its response to a write operation. Consider the following levels of conceptual write concern, listed from weakest to strongest:

Errors Ignored

With a *errors ignored* write concern, write operations are not acknowledged by MongoDB. Write operations may not succeed in the case of connection errors that the client is not yet aware of, or if the `mongod` (page 1049) produces an exception (e.g. a duplicate key exception for *unique indexes* (page 314).) While this operation is efficient because it does not require the database to respond to every write operation, it also incurs a significant risk with regards to the persistence and durability of the data.

Warning: Do not use this option in normal operation.

Unacknowledged

With a *unacknowledged* write concern, MongoDB does not acknowledge the receipt of write operation as with a write concern level of *ignore*; however, the driver will receive and handle network errors, as possible given system networking configuration.

Before the releases outlined in *Default Write Concern Change* (page 1223), this was the default write concern.

Acknowledged

With a receipt *acknowledged* write concern, the mongod (page 1049) confirms the receipt of the write operation. With this level of write concern, clients can catch network, duplicate key, and other exceptions.

This is the default write concern, after the releases outlined in *Default Write Concern Change* (page 1223).

Internally, the default write concern is to call getLastError (page 861) with no arguments. For replica sets, you can define the default write concern settings in the getLastErrorDefaults (page 477) If getLastErrorDefaults (page 477) does not define a default write concern setting, getLastError (page 861) defaults to basic receipt acknowledgment.

Journaled

With a *journaled* write concern, the mongod (page 1049) will confirm the write operation only after it has written the operation to the *journal*. This confirms that the write operation can survive a mongod (page 1049) shutdown and ensures that the write operation is durable.

While receipt *acknowledged* without *journaled* provides the fundamental basis for write concern, there is a window between journal commits where the write operation is not fully durable. See journalCommitInterval (page 1119) for more information on this window. Require *journaled* as part of the write concern to provide this durability guarantee.

Replica Acknowledged

Replica sets present an additional layer of consideration for write concern. Basic write concern levels affect the write operation on only one mongod (page 1049) instance. The w argument to getLastError (page 861) provides a *replica acknowledged* level of write concern. With *replica acknowledged* you can guarantee that the write operation has propagated to the members of a replica set. See the *Write Concern for Replica Sets* (page 396) document for more information.

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

Write Concern for Replica Sets

MongoDB's built-in *write concern* confirms the success of write operations to a *replica set's primary*. Write concern uses the getLastError (page 861) command after write operations to return an object with error information or confirmation that there are no errors.

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent. However, replica sets offer some configuration options for write and read operations. [5]

Verify Write Operations

The default write concern confirms write operations only on the primary. You can configure write concern to confirm write operations to additional replica set members as well by issuing the getLastError (page 861) command with the w option.

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

The w option confirms that write operations have replicated to the specified number of replica set members, including the primary. You can either specify a number or specify majority, which ensures the write propagates to a majority of set members.

If you specify a w value greater than the number of members that hold a copy of the data (i.e., greater than the number of non-*arbiter* members), the operation blocks until those members become available. This can cause the operation to block forever. To specify a timeout threshold for the getLastError (page 861) operation, use the wtimeout argument.

See *getLastError Examples* (page 861) for example invocations.

Modify Default Write Concern

You can configure your own "default" getLastError (page 861) behavior for a replica set. Use the getLastErrorDefaults (page 477) setting in the *replica set configuration* (page 473). The following sequence of commands creates a configuration that waits for the write operation to complete on a majority of the set members before returning:

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

The getLastErrorDefaults (page 477) setting affects only those getLastError (page 861) commands that have *no* other arguments.

Note: Use of insufficient write concern can lead to *rollbacks* (page 393) in the case of *replica set failover* (page 388). Always ensure that your operations have specified the required write concern for your application.

See also:

Write Concern (page 54) and *Write Concern Options* (page 1144)

Custom Write Concerns

You can use replica set tags to create custom write concerns using the getLastErrorDefaults (page 477) and getLastErrorModes (page 477) replica set settings.

Note: Custom write concern modes specify the field name and a number of *distinct* values for that field. By contrast, read preferences use the value of fields in the tag document to direct read operations.

In some cases, you may be able to use the same tags for read preferences and write concerns; however, you may need to create additional tags for write concerns depending on the requirements of your application.

Single Tag Write Concerns

Consider a five member replica set, where each member has one of the following tag sets:

```
{ "use": "reporting" }
{ "use": "backup" }
{ "use": "application" }
{ "use": "application" }
{ "use": "application" }
```

You could create a custom write concern mode that will ensure that applicable write operations will not return until members with two different values of the `use` tag have acknowledged the write operation. Create the mode with the following sequence of operations in the `mongo` (page 1066) shell:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { use2: { "use": 2 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `multiUse` to the `w` option of `getLastError` (page 861) as follows:

```
db.runCommand( { getLastError: 1, w: "use2" } )
```

Specific Custom Write Concerns

If you have a three member replica with the following tag sets:

```
{ "disk": "ssd" }
{ "disk": "san" }
{ "disk": "spinning" }
```

You cannot specify a custom `getLastErrorModes` (page 477) value to ensure that the write propagates to the `san` before returning. However, you may implement this write concern policy by creating the following additional tags, so that the set resembles the following:

```
{ "disk": "ssd" }
{ "disk": "san", "disk.san": "san" }
{ "disk": "spinning" }
```

Then, create a custom `getLastErrorModes` (page 477) value, as follows:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { san: { "disk.san": 1 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `san` to the `w` option of `getLastError` (page 861) as follows:

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

This operation will not return until a replica set member with the tag `disk.san` returns.

You may set a custom write concern mode as the default write concern mode using `getLastErrorDefaults` (page 477) replica set as in the following setting:

```
cfg = rs.conf()
cfg.settings.getLastErrorDefaults = { ssd: 1 }
rs.reconfig(cfg)
```

See also:

Configure Replica Set Tag Sets (page 444) for further information about replica set reconfiguration and tag sets.

28.4.2 Read Preference

Read preference describes how MongoDB clients route read operations to 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, or reduce latency, 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 388) situations where a set has *no* primary for 10 seconds or more. In this use case, you should give the application the `primaryPreferred` (page 400) 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` (page 1066) shell, see the `readPref()` method. For more information about a driver's read preference configuration, see the appropriate *MongoDB Drivers and Client Libraries* (page 575) 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* policies to ensure proper data replication and consistency.

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 495) 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 575) support five read preference modes:

- `primary` (page 400)
- `primaryPreferred` (page 400)
- `secondary` (page 400)
- `secondaryPreferred` (page 400)
- `nearest` (page 400)

You can specify a read preference mode on connection objects, database object, collection object, or per-operation. 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 1061). The `mongos` (page 1061) instance obeys specified read preferences when connecting to the *replica set* that provides each *shard* in the cluster.

In the `mongo` (page 1066) shell, the `readPref()` (page 989) cursor method provides access to read preferences.

> **Warning:** All read preference modes except `primary` (page 400) 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 400) mode.

For more information, see *read preference background* (page 399) and *read preference behavior* (page 402). 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.

The `primary` (page 400) read preference mode is not compatible with read preference modes that use *tag sets* (page 401). If you specify a tag set with `primary` (page 400), the driver will produce an error.

`primaryPreferred`

In most situations, operations read from the *primary* member of the set. However, if the primary is unavailable, as is the case during *failover* situations, operations read from secondary members.

When the read preference includes a *tag set* (page 401), 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 400) mode may return stale data in some situations.

> **Warning:** Changed in version 2.2: `mongos` (page 1061) added full support for read preferences.
> When connecting to a `mongos` (page 1061) instance older than 2.2, using a client that supports read preference modes, `primaryPreferred` (page 400) 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 401), 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 403). If no secondaries have matching tags, the read operation produces an error. [6]

Read operations using the `secondary` (page 400) 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 401), 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 403). If no secondaries have matching tags, the read operation produces an error.

Read operations using the `secondaryPreferred` (page 400) mode may return stale data.

`nearest`

The driver reads from the *nearest* member of the *set* according to the *member selection* (page 403) process.

[6] If your set has more than one secondary, and you use the `secondary` (page 400) read preference mode, consider the following effect. If you have a *three member replica set* (page 384) with a primary and two secondaries, and if one secondary becomes unavailable, all `secondary` (page 400) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.

Reads in the `nearest` (page 400) mode do not consider the member's *type*. Reads in `nearest` (page 400) 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 401), the client attempts to find a replica set member that matches the specified tag set and directs reads to an arbitrary member from among the *nearest group* (page 403).

Read operations using the `nearest` (page 400) 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 400) mode prefers low latency reads over a member's *primary* or *secondary* status.

For `nearest` (page 400), 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 403) for more information.

Tag Sets

Tag sets allow you to specify custom *read preferences* (page 398) and *write concerns* (page 54) so that your application can target operations to specific members, based on custom parameters.

Note: Consider the following properties of read preferences:

- Custom read preferences and write concerns evaluate tags sets in different ways.
- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns ignore the value of a tag to when selecting a member *except* to consider whether or not the value is unique.

You can specify tag sets with the following read preference modes:

- `primaryPreferred` (page 400)
- `secondary` (page 400)
- `secondaryPreferred` (page 400)
- `nearest` (page 400)

You cannot specify tag sets with the `primary` (page 400) read preference mode.

Tags are not compatible with `primary` (page 400) and only apply when *selecting* (page 403) a *secondary* member of a set for a read operation. However, the `nearest` (page 400) 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 403) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For information on configuring tag sets, see *Configure Replica Set Tag Sets* (page 444).

For more information on how read preference *modes* (page 399) 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` (page 1049) 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 1061):

- Reuse a connection to specific `mongod` (page 1049) for as long as possible after establishing a connection to that instance. This connection is *pinned* to this `mongod` (page 1049).
- Attempt to reconnect to a new member, obeying existing *read preference modes* (page 399), if the connection to `mongod` (page 1049) 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 399) and *tag set* (page 401). If there are fewer than three members of the set, the client will error after connecting to all existing members of the set.

  After this error, the driver selects a new member using the specified read preference mode. In the absence of a specified read preference, the driver uses `primary` (page 400).
- After detecting a failover situation, [7] the driver attempts to refresh the state of the replica set as quickly as possible.

Request Association

Important: *Request association* is configurable by the application. See your *driver* (page 575) documentation about request association configuration and default behavior.

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` (page 1049) closes connections during a *failover*. This triggers a *retry* (page 402), which may be transparent to the application.

[7] 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*.

When using request association, if the client detects that the set has elected a new primary, the driver will discard all associations between threads and read preferences.

Member Selection

Clients, by way of their drivers, and `mongos` (page 1061) instances for sharded clusters periodically update their view of the replica set's state: which members are up or down, which member is primary, and the latency to each `mongod` (page 1049) 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. Excludes members not matching the tag sets, if specified.
3. Determines which suitable member is the closest to the client in absolute terms.
4. Builds a list of members that are within a defined ping distance (in milliseconds) of the "absolute nearest" member. [8]
5. Selects a member from these hosts at random. The member receives the read operation.

Drivers can then associate this thread or connection with this member. This "*request association* (page 402)" is configurable by the application. See your *driver* (page 575) documentation about request association configuration and default behavior.

Sharding and `mongos`

Changed in version 2.2: Before version 2.2, `mongos` (page 1061) did not support the *read preference mode semantics* (page 399).

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 1061) instances that are actually connected to the set members. `mongos` (page 1061) 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 1061) is at least version 2.2. All `mongos` (page 1061) maintain their own connection pool to the replica set members. As a result:

- A request without a specified preference has `primary` (page 400), the default, unless, the `mongos` (page 1061) reuses an existing connection that has a different mode set.

  Always explicitly set your read preference mode to prevent confusion.

- All `nearest` (page 400) and latency calculations reflect the connection between the `mongos` (page 1061) and the `mongod` (page 1049) instances, not the client and the `mongod` (page 1049) instances.

  This produces the desired result, because all results must pass through the `mongos` (page 1061) before returning to the client.

[8] Applications can configure the threshold used in this stage. The default "acceptable latency" is 15 milliseconds, which you can override in the drivers with their own `secondaryAcceptableLatencyMS` option. For `mongos` (page 1061) you can use the `--localThreshold` or `localThreshold` (page 1126) runtime options to set this value.

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 399) for the following commands:

- `group` (page 836)
- `mapReduce` (page 840) [9]
- `aggregate` (page 834)
- `collStats` (page 900)
- `dbStats` (page 904)
- `count` (page 834)
- `distinct` (page 835)
- `geoNear` (page 850)
- `geoSearch` (page 851)
- `geoWalk` (page 851)

New in version 2.4: `mongos` (page 1061) adds support for routing commands to shards using read preferences. Previously `mongos` (page 1061) sent all commands to shards' primaries.

Uses for non-Primary Read Preferences

You must exercise care when specifying read preferences: modes other than `primary` (page 400) can *and will* return stale data. These secondary queries will not include the most recent write operations to the replica set's *primary*. Nevertheless, there are several common use cases for using non-`primary` (page 400) read preference modes:

- Reporting and analytics workloads.

  Issuing reads to secondaries for some work profiles helps distribute load and prevent operations from affecting the main workload of the primary.

  Also consider using `secondary` (page 400) in conjunction with a direct connection to a *hidden member* (page 379) of the set.

  Note: Read preferences aren't relevant to direct connections to a single `mongod` (page 1049) instance. However, in order to perform read operations on a direct connection to a secondary member of a replica set, you must set a read preference, such as *secondary*.

- Providing local reads for geographically distributed applications.

  If you have application servers in multiple data centers, you may consider having a *geographically distributed replica set* (page 388) and using a non primary read preference or the `nearest` (page 400) to avoid network latency.

- Maintaining availability during a failover.

  Use `primaryPreferred` (page 400) if you want your application to do consistent reads from the primary under normal circumstances, but to allow stale reads from secondaries in an emergency. This provides a "read-only mode" for your application during a failover.

[9] Only "inline" `mapReduce` (page 840) operations that do not write data support read preference, otherwise these operations must run on the *primary* members.

Warning: In some situations using `secondaryPreferred` (page 400) to distribute read load to replica sets may carry significant operational risk: if all secondaries are unavailable and your set has enough *arbiters* to prevent the primary from stepping down, then the primary will receive all traffic from clients.
For this reason, use `secondary` (page 400) to distribute read load to replica sets, not `secondaryPreferred` (page 400).

Using read modes other than `primary` (page 400) and `primaryPreferred` (page 400) to provide extra capacity is not in and of itself justification for non-`primary` (page 400) in many cases. Furthermore, *sharding* (page 485) increases read and write capacity by distributing read and write operations across a group of machines.

28.5 Replication Processes

Members of a *replica set* replicate data continuously. First, members use *initial sync* to capture the data set. Then, every `mongod` (page 1049) instance keeps an *oplog* (page 405), which is a *capped collection*. to support replication. The oplog records every operation that modifies the data set.

***Replica Set Oplog* (page 405)** The oplog records all operations that modify the data in the replica set.

***Replica Set Data Synchronization* (page 406)** Secondaries must replicate all changes accepted by the primary. This process is the basis of replica set operations.

28.5.1 Replica Set Oplog

The *oplog* (operations log) is a special *capped collection* that keeps a rolling record of all operations that modify the data stored in your databases. MongoDB applies database operations on the *primary* and then records the operations on the primary's oplog. The *secondary* members then copy and apply these operations in an asynchronous process. All replica set members contain a copy of the oplog, allowing them to maintain the current state of the database.

To facilitate replication, all replica set members send heartbeats (pings) to all other members. Any member can import oplog entries from any other member.

Whether applied once or multiple times to the target dataset, each operation in the oplog produces the same results, i.e. each operation in the oplog is *idempotent*. For proper replication operations, entries in the oplog must be idempotent:

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

Oplog Size

When you start a replica set member for the first time, MongoDB creates an oplog of a default size. The size depends on the architectural details of your operating system.

In most cases, the default oplog size is sufficient. For example, if an oplog is 5% of free disk space and fills up in 24 hours of operations, then secondaries can stop copying entries from the oplog for up to 24 hours without becoming stale. However, most replica sets have much lower operation volumes, and their oplogs can hold much higher numbers of operations.

Before `mongod` (page 1049) creates an oplog, you can specify its size with the `oplogSize` (page 1124) option. However, after you have started a replica set member for the first time, you can only change the size of the oplog using the *Change the Size of the Oplog* (page 439) procedure.

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

- For 64-bit Linux, Solaris, FreeBSD, and Windows systems, MongoDB allocates 5% of the available free disk space to the oplog. If this amount is smaller than a gigabyte, then MongoDB allocates 1 gigabyte of space.
- For 64-bit OS X systems, MongoDB allocates 183 megabytes of space to the oplog.
- For 32-bit systems, MongoDB allocates about 48 megabytes of space to the oplog.

Workloads that Might Require a Larger Oplog Size

If you can predict your replica set's workload to resemble one of the following patterns, then you might want to create an oplog that is larger than the default. Conversely, if your application predominantly performs reads and writes only a small amount of data, you will oplog may be sufficient.

The following workloads might require a larger oplog size.

Updates to Multiple Documents at Once

The oplog must translate multi-updates into individual operations in order to maintain *idempotency*. This can use a great deal of oplog space without a corresponding increase in data size or disk use.

Deletions Equal the Same Amount of Data as Inserts

If you delete roughly the same amount of data as you insert, the database will not grow significantly in disk use, but the size of the operation log can be quite large.

Significant Number of In-Place Updates

If a significant portion of the workload is in-place updates, the database records a large number of operations but does not change the quantity of data on disk.

Oplog Status

To view oplog status, including the size and the time range of operations, issue the `db.printReplicationInfo()` (page 1010) method. For more information on oplog status, see *Check the Size of the Oplog* (page 457).

Under various exceptional situations, updates to a *secondary's* oplog might lag behind the desired performance time. Use `db.getReplicationInfo()` (page 1006) from a secondary member and the *replication status* (page 1006) output to assess the current state of replication and determine if there is any unintended replication delay.

See *Replication Lag* (page 455) for more information.

28.5.2 Replica Set Data Synchronization

In order to maintain up to date copies of the shared data set, members of a replica set *sync* or replicate data from other members. MongoDB uses two forms of data synchronization: *initial sync* (page 407) to populate new members with the full data set, and replication to apply ongoing changes to the entire data set.

Initial Sync

Replica set members use initial sync in two situations: when a member has no data and when the member has data but is missing a history of the replica set's replication. The `mongod` (page 1049) instance uses the initial sync to copy all data from another member in the replica set. The procedure for an initial sync is:

1. Clone all databases. To clone, the `mongod` (page 1049) queries every collection in each source database, and inserts all data into its own copies of these collections.
2. Applies all changes to the data set. Using the oplog from the source, the `mongod` (page 1049) updates its data set to reflect the current state of the replica set.
3. Builds all indexes on all collections.

When the `mongod` (page 1049) completes all index builds, the member can transition to a normal state, such as *secondary*.

Replication

Replica set members replicate data continuously after the initial sync. This process keeps the members up to date with all changes to the replica set's data. In most cases, secondaries synchronize from the primary. Secondaries may automatically change their *sync targets* if needed based on changes in the ping time and state of other members' replication.

For a member to sync from another, the `buildIndexes` (page 475) setting for both members must have the same value/ `buildIndexes` (page 475) must be either `true` or `false` for both members.

Beginning in version 2.2, secondaries avoid syncing from *delayed members* (page 379) and *hidden members* (page 379).

Consistency and Durability

In a replica set, only the primary can accept write operations. Writing only to the primary provides *strict consistency* among members.

Journaling provides single-instance write durability. Without journaling, if a MongoDB instance terminates ungracefully, you should assume that the database is in a corrupt or inconsistent state.

Multithreaded Replication

MongoDB applies write operations in batches using multiple threads to improve concurrency. MongoDB groups batches by namespace and applies operations using a group of threads, but always applies the write operations to a namespace in order.

While applying a batch, MongoDB blocks all reads. As a result, secondaries can never return data that reflects a state that never existed on the primary.

Pre-Fetching Indexes to Improve Replication Throughput

To help improve the performance of applying oplog entries, MongoDB fetches memory pages that hold affected data and indexes. This *pre-fetch* stage minimizes the amount of time MongoDB holds the write lock while applying oplog entries. By default, secondaries will pre-fetch all *Indexes* (page 307).

Optionally, you can disable all pre-fetching or only pre-fetch the index on the `_id` field. See the `replIndexPrefetch` (page 1124) setting for more information.

28.6 Master Slave Replication

Important: *Replica sets* (page 373) replace *master-slave* replication for most use cases. If possible, use replica sets rather than master-slave replication for all new production deployments This documentation remains to support legacy deployments and for archival purposes, *only*.

In addition to providing all the functionality of master-slave deployments, replica sets are also more robust for production use. Master-slave replication preceded replica and makes it possible have a large number of non-master (i.e. slave) and to *only* replicate operations for a single database; however, master-slave replication provides less redundancy, and does not automate failover. See *Deploy Master-Slave Equivalent using Replica Sets* (page 410) for a replica set configuration that is equivalent to master-slave replication. If you wish to convert an existing master-slave deployment to a replica set, see *Convert a Master-Slave Deployment to a Replica Set* (page 411).

28.6.1 Fundamental Operations

Initial Deployment

To configure a *master-slave* deployment, start two `mongod` (page 1049) instances: one in `master` (page 1125) mode, and the other in `slave` (page 1125) mode.

To start a `mongod` (page 1049) instance in `master` (page 1125) mode, invoke `mongod` (page 1049) as follows:

```
mongod --master --dbpath /data/masterdb/
```

With the `--master` option, the `mongod` (page 1049) will create a `local.oplog.$main` (page 480) collection, which the "operation log" that queues operations that the slaves will apply to replicate operations from the master. The `--dbpath` is optional.

To start a `mongod` (page 1049) instance in `slave` (page 1125) mode, invoke `mongod` (page 1049) as follows:

```
mongod --slave --source <masterhostname><:<port>> --dbpath /data/slavedb/
```

Specify the hostname and port of the master instance to the `--source` argument. The `--dbpath` is optional.

For `slave` (page 1125) instances, MongoDB stores data about the source server in the `local.sources` (page 480) collection.

Configuration Options for Master-Slave Deployments

As an alternative to specifying the `--source` run-time option, can add a document to `local.sources` (page 480) specifying the `master` (page 1125) instance, as in the following operation in the `mongo` (page 1066) shell:

```
1 use local
2 db.sources.find()
3 db.sources.insert( { host: <masterhostname> <,only: databasename> } );
```

In line 1, you switch context to the `local` database. In line 2, the `find()` (page 951) operation should return no documents, to ensure that there are no documents in the `sources` collection. Finally, line 3 uses `db.collection.insert()` (page 961) to insert the source document into the `local.sources` (page 480) collection. The model of the `local.sources` (page 480) document is as follows:

host
The host field specifies the `master` (page 1125)`mongod` (page 1049) instance, and holds a resolvable hostname, i.e. IP address, or a name from a `host` file, or preferably a fully qualified domain name.

You can append <:port> to the host name if the mongod (page 1049) is not running on the default 27017 port.

only

Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

Operational Considerations for Replication with Master Slave Deployments

Master instances store operations in an *oplog* which is a *capped collection* (page 578). As a result, if a slave falls too far behind the state of the master, it cannot "catchup" and must re-sync from scratch. Slave may become out of sync with a master if:

- The slave falls far behind the data updates available from that master.
- The slave stops (i.e. shuts down) and restarts later after the master has overwritten the relevant operations from the master.

When slaves, are out of sync, replication stops. Administrators must intervene manually to restart replication. Use the resync (page 870) command. Alternatively, the *--autoresync* allows a slave to restart replication automatically, after ten second pause, when the slave falls out of sync with the master. With *--autoresync* specified, the slave will only attempt to re-sync once in a ten minute period.

To prevent these situations you should specify a larger oplog when you start the master (page 1125) instance, by adding the *--oplogSize* option when starting mongod (page 1049). If you do not specify *--oplogSize*, mongod (page 1049) will allocate 5% of available disk space on start up to the oplog, with a minimum of 1GB for 64bit machines and 50MB for 32bit machines.

28.6.2 Run time Master-Slave Configuration

MongoDB provides a number of run time configuration options for mongod (page 1049) instances in *master*-*slave* deployments. You can specify these options in *configuration files* (page 129) or on the command-line. See documentation of the following:

- For *master* nodes:
  - master (page 1125)
  - slave (page 1125)
- For *slave* nodes:
  - source (page 1125)
  - only (page 1125)
  - slaveDelay (page 1125)

Also consider the *Master-Slave Replication Command Line Options* (page 1058) for related options.

Diagnostics

On a *master* instance, issue the following operation in the mongo (page 1066) shell to return replication status from the perspective of the master:

```
db.printReplicationInfo()
```

On a *slave* instance, use the following operation in the mongo (page 1066) shell to return the replication status from the perspective of the slave:

```
db.printSlaveReplicationInfo()
```

Use the `serverStatus` (page 919) as in the following operation, to return status of the replication:

```
db.serverStatus()
```

See *server status repl fields* (page 927) for documentation of the relevant section of output.

28.6.3 Security

When running with `auth` (page 1118) enabled, in *master-slave* deployments configure a `keyFile` (page 1117) so that slave `mongod` (page 1049) instances can authenticate and communicate with the master `mongod` (page 1049) instance.

To enable authentication and configure the `keyFile` (page 1117) add the following option to your configuration file:

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

Note: You may chose to set these run-time configuration options using the `--keyFile` option on the command line.

Setting `keyFile` (page 1117) enables authentication and specifies a key file for the `mongod` (page 1049) instances to use when authenticating to each other. The content of the key file is arbitrary but must be the same on all members of the deployment can connect to each other.

The key file must be less one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or "world" permissions on UNIX systems. Use the following command to use the OpenSSL package to generate "random" content for use in a key file:

```
openssl rand -base64 741
```

See also:

Security (page 205) for more information about security in MongoDB

28.6.4 Ongoing Administration and Operation of Master-Slave Deployments

Deploy Master-Slave Equivalent using Replica Sets

If you want a replication configuration that resembles *master-slave* replication, using *replica sets* replica sets, consider the following replica configuration document. In this deployment hosts `<master>` and `<slave>` [10] provide replication that is roughly equivalent to a two-instance master-slave deployment:

```
{
  _id : 'setName',
  members : [
    { _id : 0, host : "<master>", priority : 1 },
    { _id : 1, host : "<slave>", priority : 0, votes : 0 }
  ]
}
```

See *Replica Set Configuration* (page 473) for more information about replica set configurations.

[10] In replica set configurations, the `host` (page 474) field must hold a resolvable hostname.

Convert a Master-Slave Deployment to a Replica Set

To convert a master-slave deployment to a replica set, restart the current master as a one-member replica set. Then remove the data directors from previous secondaries and add them as new secondaries to the new replica set.

1. To confirm that the current instance is master, run:

   ```
   db.isMaster()
   ```

   This should return a document that resembles the following:

   ```
   {
           "ismaster" : true,
           "maxBsonObjectSize" : 16777216,
           "maxMessageSizeBytes" : 48000000,
           "localTime" : ISODate("2013-07-08T20:15:13.664Z"),
           "ok" : 1
   }
   ```

2. Shut down the `mongod` (page 1049) processes on the master and all slave(s), using the following command while connected to each instance:

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

3. Back up your `http://docs.mongodb.org/manualdata/db` directories, in case you need to revert to the master-slave deployment.

4. Start the former master with the *--replSet* option, as in the following:

   ```
   mongod --replSet <setname>
   ```

5. Connect to the `mongod` (page 1049) with the `mongo` (page 1066) shell, and initiate the replica set with the following command:

   ```
   rs.initiate()
   ```

   When the command returns, you will have successfully deployed a one-member replica set. You can check the status of your replica set at any time by running the following command:

   ```
   rs.status()
   ```

You can now follow the *convert a standalone to a replica set* (page 426) tutorial to deploy your replica set, picking up from the *Expand the Replica Set* (page 427) section.

Failing over to a Slave (Promotion)

To permanently failover from a unavailable or damaged *master* (`A` in the following example) to a *slave* (`B`):

1. Shut down `A`.

2. Stop `mongod` (page 1049) on `B`.

3. Back up and move all data files that begin with `local` on `B` from the `dbpath` (page 1118).

   > **Warning:** Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

4. Restart `mongod` (page 1049) on `B` with the *--master* option.

Note: This is a one time operation, and is not reversible. `A` cannot become a slave of `B` until it completes a full resync.

Inverting Master and Slave

If you have a *master* (`A`) and a *slave* (`B`) and you would like to reverse their roles, follow this procedure. The procedure assumes `A` is healthy, up-to-date and available.

If `A` is not healthy but the hardware is okay (power outage, server crash, etc.), skip steps 1 and 2 and in step 8 replace all of `A`'s files with `B`'s files in step 8.

If `A` is not healthy and the hardware is not okay, replace `A` with a new machine. Also follow the instructions in the previous paragraph.

To invert the master and slave in a deployment:

1. Halt writes on `A` using the *fsync* command.
2. Make sure `B` is up to date with the state of `A`.
3. Shut down `B`.
4. Back up and move all data files that begin with `local` on `B` from the `dbpath` (page 1118) to remove the existing `local.sources` data.

   Warning: Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

5. Start `B` with the *--master* option.
6. Do a write on `B`, which primes the *oplog* to provide a new sync start point.
7. Shut down `B`. `B` will now have a new set of data files that start with `local`.
8. Shut down `A` and replace all files in the `dbpath` (page 1118) of `A` that start with `local` with a copy of the files in the `dbpath` (page 1118) of `B` that begin with `local`.

   Considering compressing the `local` files from `B` while you copy them, as they may be quite large.

9. Start `B` with the *--master* option.
10. Start `A` with all the usual slave options, but include *fastsync*.

Creating a Slave from an Existing Master's Disk Image

If you can stop write operations to the *master* for an indefinite period, you can copy the data files from the master to the new *slave* and then start the slave with *--fastsync*.

Warning: Be careful with *--fastsync*. If the data on both instances is identical, a discrepancy will exist forever.

`fastsync` (page 1124) is a way to start a slave by starting with an existing master disk image/backup. This option declares that the administrator guarantees the image is correct and completely up-to-date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a Slave from an Existing Slave's Disk Image

You can just copy the other *slave's* data file snapshot without any special options. Only take data snapshots when a `mongod` (page 1049) process is down or locked using `db.fsyncLock()` (page 1004).

Resyncing a Slave that is too Stale to Recover

Slaves asynchronously apply write operations from the *master* that the slaves poll from the master's *oplog*. The oplog is finite in length, and if a slave is too far behind, a full resync will be necessary. To resync the slave, connect to a slave using the `mongo` (page 1066) and issue the `resync` (page 870) command:

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

This forces a full resync of all data (which will be very slow on a large database). You can achieve the same effect by stopping `mongod` (page 1049) on the slave, deleting the entire content of the `dbpath` (page 1118) on the slave, and restarting the `mongod` (page 1049).

Slave Chaining

Slaves cannot be "chained." They must all connect to the *master* directly.

If a slave attempts "slave from" another slave you will see the following line in the `mongod` (page 1049) long of the shell:

```
assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main
```

Correcting a Slave's Source

To change a *slave's* source, manually modify the slave's `local.sources` (page 480) collection.

Example

Consider the following: If you accidentally set an incorrect hostname for the slave's `source` (page 1125), as in the following example:

```
mongod --slave --source prod.mississippi
```

You can correct this, by restarting the slave without the `--slave` and `--source` arguments:

```
mongod
```

Connect to this `mongod` (page 1049) instance using the `mongo` (page 1066) shell and update the `local.sources` (page 480) collection, with the following operation sequence:

```
use local

db.sources.update( { host : "prod.mississippi" },
                   { $set : { host : "prod.mississippi.example.net" } } )
```

Restart the slave with the correct command line arguments or with no `--source` option. After configuring `local.sources` (page 480) the first time, the `--source` will have no subsequent effect. Therefore, both of the following invocations are correct:

```
mongod --slave --source prod.mississippi.example.net
```

or

```
mongod --slave
```

The slave now polls data from the correct *master*.

CHAPTER 29

Replica Set Tutorials

The administration of *replica sets* includes the initial deployment of the set, adding and removing members to a set, and configuring the operational parameters and properties of the set. Administrators generally need not intervene in failover or replication processes as MongoDB automates these functions. In the exceptional situations that require manual interventions, the tutorials in these sections describe processes such as resyncing a member. The tutorials in this section form the basis for all replica set administration.

***Troubleshoot Replica Sets* (page 455)** Describes common issues and operational challenges for replica sets. For additional diagnostic information, see *FAQ: MongoDB Diagnostics* (page 777).

29.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.

29.1.1 Deploy a Replica Set

This tutorial describes how to create a three-member *replica set* from three existing `mongod` (page 1049) instances. The tutorial provides two procedures: one for development and test systems; and a one for production systems.

To instead deploy a replica set from a single standalone MongoDB instance, see *Convert a Standalone to a Replica Set* (page 426). For additional information regarding replica set deployments, see *Replication* (page 367) and *Replica Set Deployment Architectures* (page 382).

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.

Requirements

Most replica sets consist of three or more `mongod` (page 1049) instances. [1] This tutorial describes a three member set. Production environments should have at least three distinct systems so that each system can run its own instance of `mongod` (page 1049). For development systems you can run all three instances of the `mongod` (page 1049) process on a local system or within a virtual instance. For production environments, you should maintain as much separation between members as possible. For example, when using virtual machines for production deployments, each member should live on a separate host server, served by redundant power circuits and with redundant network paths.

[1] To ensure smooth *elections* (page 389) always design replica sets with odd numbers of members. Use *Arbiter* (page **??**) to ensure the set has odd number of voting members and avoid tied elections.

Procedures

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 3).

Deploy a Development or Test 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 two members. To test connectivity, see *Test Connections Between all Members* (page 457).

2. Start three instances of `mongod` (page 1049) 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` (page 1049) instances in separate windows of GNU Screen. OS X and most Linux distributions come with screen installed by default [2] 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 --smallfiles --oplogSize 128
   mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0 --smallfiles --oplogSize 128
   mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0 --smallfiles --oplogSize 128
   ```

   This starts each instance as a member of a replica set named `rs0`, each running on a distinct port. If you are already using these ports, select different ports.

   The `--smallfiles` and `--oplogSize` reduce the disk space that each `mongod` (page 1049) uses for the purposes of testing. For more information on these and other configuration options, see *Configuration File Options* (page 1115).

   Important: Unless otherwise indicated, only use these invocations for test deployments.

3. Open a `mongo` (page 1066) shell and connect to the first `mongod` (page 1049) instance, with the following command:

   ```
   mongo --port 27017
   ```

4. Create a replica set configuration object in the `mongo` (page 1066) shell environment to use to initiate the replica set with the following sequence of operations:

   ```
   rsconf = {
              _id: "rs0",
              members: [
                         {
                          _id: 0,
                          host: "<hostname>:27017"
                         }
                       ]
            }
   ```

5. Use `rs.initiate()` (page 1016) to initiate a replica set consisting of the current member and using the default configuration:

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

```
rs.initiate( rsconf )
```

6. Display the current *replica configuration* (page 473):

```
rs.conf()
```

7. Add the second and third `mongod` (page 1049) instances to the replica set using the `rs.add()` (page 1014) method. Replace `<hostname>` with your system's hostname in the following examples:

```
rs.add("<hostname>:27018")
rs.add("<hostname>:27019")
```

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

8. Check the status of your replica set at any time with the `rs.status()` (page 1017) operation.

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 1016)
- `rs.conf()` (page 1015)
- `rs.reconfig()` (page 1016)
- `rs.add()` (page 1014)

You may also consider the simple setup script as an example of a basic automatically configured replica set.

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' `/etc/hosts` file to reflect this configuration.
- You specify run-time configuration on each system in a *configuration file* (page 1115) stored in `/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 1116). However, if you do not specify an interface, MongoDB listens for connections on all available IPv4 interfaces. Modify bind_ip (page 1116) 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 1115).

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 two members. To test connectivity, see *Test Connections Between all Members* (page 457).

2. On each system start the mongod (page 1049) 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 (page 1066) shell connected to this host:

   ```
   mongo
   ```

4. Use rs.initiate() (page 1016) to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

5. Display the current *replica configuration* (page 473):

   ```
   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 few seconds.

7. Check the status of your replica set at any time with the rs.status() (page 1017) operation.

See also:

The documentation of the following shell functions for more information:

- rs.initiate() (page 1016)
- rs.conf() (page 1015)
- rs.reconfig() (page 1016)
- rs.add() (page 1014)

29.1.2 Deploy a Geographically Distributed Replica Set

This tutorial outlines the process for deploying a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

For appropriate background, see *Replication* (page 367) and *Replica Set Deployment Architectures* (page 382). For related tutorials, see *Deploy a Replica Set* (page 416) and *Add Members to a Replica Set* (page 427).

Overview

While *replica sets* provide basic protection against single-instance failure, when all of the members of a replica set reside in a single facility, the replica set is still susceptible to some classes of errors in 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.

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 392) are within Site A. This includes *secondary-only members* (page 378) and *arbiters* (page **??**) For more information on the need to keep the voting majority on one site, see *Replica Set Elections* (page 389).
- If you deploy a replica set with an uneven number of members, deploy an *arbiter* (page **??**) 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.

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' `/etc/hosts` 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 routes all traffic between members within a single site over the local area network.
  - Configure authentication using `auth` (page 1118) and `keyFile` (page 1117), 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 *Inter-Process Authentication* (page 212).
- Specify run-time configuration on each system in a *configuration file* (page 1115) stored in `/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 1116) 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 1124) following the `http://docs.mongodb.org/manual` provides a "seed list" of known members of the replica set. `mongod` (page 1049) uses this list to fetch configuration changes following restarts. It is acceptable to omit this section entirely, and have the `replSet` (page 1124) option resemble:

  ```
  replSet = rs0
  ```

  For more documentation on the above run time configurations, as well as additional configuration options, see *Configuration File Options* (page 1115).

To deploy a geographically distributed three-member set:

1. On each system start the `mongod` (page 1049) 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 (page 1066) shell connected to *one* of the mongod (page 1049) instances:

```
mongo
```

3. Use the rs.initiate() (page 1016) method on *one* member to initiate a replica set consisting of the current member and using the default configuration:

```
rs.initiate()
```

4. Display the current *replica configuration* (page 473):

```
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 378):

   (a) Issue the following command to determine the _id (page 474) value for mongodb2.example.net:

   ```
   rs.conf()
   ```

   (b) In the members (page 474) array, save the _id (page 474) value. The example in the next step assumes this value is 2.

   (c) In the mongo (page 1066) 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() (page 1016) shell method 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() (page 1017).

See also:

The documentation of the following shell functions for more information:

- rs.initiate() (page 1016)
- rs.conf() (page 1015)
- rs.reconfig() (page 1016)
- rs.add() (page 1014)

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' `/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 378) 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 least 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 routes all traffic between members within a single site over the local area network.
  - Configure authentication using `auth` (page 1118) and `keyFile` (page 1117), 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 *Inter-Process Authentication* (page 212).
- Specify run-time configuration on each system in a *configuration file* (page 1115) stored in `/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 1116) 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 1124) following the `http://docs.mongodb.org/manual` provides a "seed list" of known members of the replica set. `mongod` (page 1049) uses this list to fetch configuration changes following restarts. It is acceptable to omit this section entirely, and have the `replSet` (page 1124) 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` (page 1049) 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` (page 1066) shell connected to this host:

```
mongo
```

3. Use `rs.initiate()` (page 1016) to initiate a replica set consisting of the current member and using the default configuration:

```
rs.initiate()
```

4. Display the current *replica configuration* (page 473):

```
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 378):

(a) Issue the following command to determine the `_id` (page 474) value for the member:

```
rs.conf()
```

(b) In the `members` (page 474) array, save the `_id` (page 474) value. The example in the next step assumes this value is `2`.

(c) In the `mongo` (page 1066) 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()` (page 1016) shell method 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()` (page 1017).

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 1016)
- `rs.conf()` (page 1015)
- `rs.reconfig()` (page 1016)
- `rs.add()` (page 1014)

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 422)). 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 420)). Ensure that Site A always has a majority of the members of the set. For example, if a set has five members, deploy three members within the primary facility and two members in other facilities.
- If you have a majority of the members of the set *outside* of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.

29.1.3 Add an Arbiter to Replica Set

Arbiters are `mongod` (page 1049) instances that are part of *replica set* but do not hold data. Arbiters participate in *elections* (page 389) in order to break ties. If a replica set has an even number of members, add an arbiter.

Arbiters have minimal resource requirements and do not require dedicated hardware. You can deploy an arbiter on an application server, monitoring host.

Important: Do not run an arbiter on the same system as a member of the replica set.

Add an Arbiter

1. Create a data directory (e.g. `dbpath` (page 1118)) for the arbiter. The `mongod` (page 1049) instance uses the directory for configuration data. The directory *will not* hold the data set. For example, create the `http://docs.mongodb.org/manualdata/arb` directory:

   ```
   mkdir /data/arb
   ```

2. Start the arbiter. Specify the data directory and the replica set name. The following, starts an arbiter using the `http://docs.mongodb.org/manualdata/arb` `dbpath` (page 1118) for the `rs` replica set:

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

3. Connect to the primary and add the arbiter to the replica set. Use the `rs.addArb()` (page 1015) method, as in the following example:

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

   This operation adds the arbiter running on port `30000` on the `m1.example.net` host.

29.1.4 Convert a Standalone to a Replica Set

This tutorial describes the process for converting a *standalone* `mongod` (page 1049) instance into a three-member *replica set*. Use standalone instances for testing and development, but always use replica sets in production. To install a standalone instance, see the *installation tutorials* (page 3).

To deploy a replica set without using a pre-existing `mongod` (page 1049) instance, see *Deploy a Replica Set* (page 416).

Procedure

1. Shut down the *standalone* `mongod` (page 1049) instance.
2. Restart the instance. Use the `--replSet` option to specify the name of the new replica set.

   For example, the following command starts a standalone instance as a member of a new replica set named `rs0`. The command uses the standalone's existing database path of `http://docs.mongodb.org/manualsrv/mongodb/db0`:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   For more information on configuration options, see *Configuration File Options* (page 1115) and the `mongod` (page 1049) manual page.

3. Connect to the `mongod` (page 1049) instance.
4. Use `rs.initiate()` (page 1016) to initiate the new replica set:

   ```
   rs.initiate()
   ```

The replica set is now operational.

To view the replica set configuration, use `rs.conf()` (page 1015). To check the status of the replica set, use `rs.status()` (page 1017).

Expand the Replica Set

Add additional replica set members by doing the following:

1. On two distinct systems, start two new standalone `mongod` (page 1049) instances. For information on starting a standalone instance, see the *installation tutorial* (page 3) specific to your environment.
2. On your connection to the original `mongod` (page 1049) instance (the former standalone instance), issue a command in the following form for each new instance to add to the replica set:

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

   Replace <hostname> and <port> with the resolvable hostname and port of the `mongod` (page 1049) instance to add to the set. For more information on adding a host to a replica set, see *Add Members to a Replica Set* (page 427).

Sharding Considerations

If the new replica set is part of a *sharded cluster*, change the shard host information in the *config database* by doing the following:

1. Connect to one of the sharded cluster's `mongos` (page 1061) instances and issue a command in the following form:

```
db.getSiblingDB("config").shards.save( {_id: "<name>", host: "<replica-set>/<member,><member,><.
```

   Replace <name> with the name of the shard. Replace <replica-set> with the name of the replica set. Replace <member,><member,><> with the list of the members of the replica set.

2. Restart all `mongos` (page 1061) instances. If possible, restart all components of the replica sets (i.e., all `mongos` (page 1061) and all shard `mongod` (page 1049) instances).

29.1.5 Add Members to a Replica Set

Overview

This tutorial explains how to add an additional member to an existing *replica set*. For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 382) document.

Maximum Voting Members

A replica set can have a maximum of seven *voting members* (page 389). To add a member to a replica set that already has seven votes, you must either add the member as a *non-voting member* (page 392) or remove a vote from an `existing member` (page 476).

Control Scripts

In production deployments you can configure a *control script* to manage member processes.

Existing Members

You can use these procedures to add new members to an existing set. You can also use the same procedure to "re-add" a removed member. If the removed member's data is still relatively recent, it can recover and catch up easily.

Data Files

If you have a backup or snapshot of an existing member, you can move the data files (e.g. the `dbpath` (page 1118) directory) to a new system and use them to quickly initiate a new member. The files must be:

- A consistent copy of the database from a member of the same replica set. See *Use Filesystem Snapshots to Backup and Restore MongoDB Databases* (page 138) document for more information.

  Important: Always use filesystem snapshots to create a copy a member of the existing replica set. **Do not** use `mongodump` (page 1075) and `mongorestore` (page 1079) to seed a new replica set member.

- More recent than the oldest operation in the *primary's oplog*. The new member must be able to become current by applying operations from the primary's oplog.

Requirements

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

Otherwise, use the MongoDB *installation tutorial* (page 3) and the *Deploy a Replica Set* (page 416) tutorials.

Procedures

Prepare the Data Directory

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

- Make sure the new member's data directory *does not* contain data. The new member will copy the data from an existing member.

  If the new member is in a *recovering* state, it must exit and become a *secondary* before MongoDB can copy all data as part of the replication process. This process takes time but does not require administrator intervention.

- Manually copy the data directory from an existing member. The new member becomes a secondary member and will catch up to the current state of the replica set. Copying the data over may shorten the amount of time for the new member to become current.

  Ensure that you can copy the data directory to the new member and begin replication within the *window allowed by the oplog* (page 405). Otherwise, the new instance will have to perform an initial sync, which completely resynchronizes the data, as described in *Resync a Member of a Replica Set* (page 443).

  Use `db.printReplicationInfo()` (page 1010) to check the current state of replica set members with regards to the oplog.

For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 382) document.

Add a Member to an Existing Replica Set

1. Start the new `mongod` (page 1049) instance. Specify the data directory and the replica set name. The following example specifies the `http://docs.mongodb.org/manualsrv/mongodb/db0` data directory and the `rs0` replica set:

   ```
   mongod --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   Take note of the host name and port information for the new `mongod` (page 1049) instance.

   For more information on configuration options, see the `mongod` (page 1049) manual page.

   Optional

   You can specify the data directory and replica set in the `mongo.conf` *configuration file* (page 1115), and start the `mongod` (page 1049) with the following command:

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

   ---

2. Connect to the replica set's primary.

   You can only add members while connected to the primary. If you do not know which member is the primary, log into any member of the replica set and issue the `db.isMaster()` (page 1008) command.

3. Use `rs.add()` (page 1014) to add the new member to the replica set. For example, to add a member at host `mongodb3.example.net`, issue the following command:

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

   You can include the port number, depending on your setup:

   ```
   rs.add("mongodb3.example.net:27017")
   ```

4. Verify that the member is now part of the replica set. Call the `rs.conf()` (page 1015) method, which displays the *replica set configuration* (page 473):

   ```
   rs.conf()
   ```

   To view replica set status, issue the `rs.status()` (page 1017) method. For a description of the status fields, see *replSetGetStatus* (page 865).

Configure and Add a Member

You can add a member to a replica set by passing to the `rs.add()` (page 1014) method a `members` (page 474) document. The document must be in the form of a `local.system.replset.members` (page 474) document. These documents define a replica set member in the same form as the *replica set configuration document* (page 473).

Important: Specify a value for the `_id` field of the `members` (page 474) document. MongoDB does not automatically populate the `_id` field in this case. Finally, the `members` (page 474) document must declare the `host` value. All other fields are optional.

Example

To add a member with the following configuration:

- an `_id` of `1`.
- a `hostname and port number` (page 474) of `mongodb3.example.net:27017`.

- a `priority` (page 475) value within the replica set of `0`.
- a configuration as `hidden` (page 475),

Issue the following:

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

29.1.6 Remove Members from Replica Set

Use this procedure to remove or decommission a member of a *replica set*. To change the hostname of a member of the replica set, you can reconfigure the set: see *Change Hostnames in a Replica Set* (page 451) for more information..

Requirements

For best results always *shut down* the `mongod` (page 1049) instance before removing it from a *replica set*.

Changed in version 2.2: In versions prior to 2.2, you *must* to shut down the `mongod` (page 1049) instance before removing it.

Procedure

1. Connect to the *primary* with the `mongo` (page 1066) shell.

   Use `db.isMaster()` (page 1008) when connected to *any* member of the set to find the current primary.

2. Use `rs.remove()` (page 1017) in either of the following forms to remove the member:

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

   MongoDB disconnects the shell briefly as the replica set elects a new primary: the shell will automatically reconnect. However, the shell displays an error even if this command succeeds.

To add a removed member to a replica set, see the *procedure for adding replica set members* (page 427).

29.1.7 Replace a Replica Set Member

If you need to change the hostname of a replica set member without changing the configuration of that member or the set, you can use the operation outlined in this tutorial. For example if you must re-provision systems or rename hosts, you can use this pattern to minimize the scope of that change.

Operation

To change the hostname for a replica set member modify the `host` (page 474) field. The value of `_id` (page 474) field will not change when you reconfigure the set.

See *Replica Set Configuration* (page 473) and `rs.reconfig()` (page 1016) for more information.

Note: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 389). During the election, the current shell session and clients connected to this replica set disconnect, which produces an error even when the operation succeeds.

Example

To change the hostname to mongo2.example.net for the replica set member configured at members[0], issue the following sequence of commands:

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

29.2 Member Configuration Tutorials

The following tutorials provide information in configuring replica set members to support specific operations, such as to provide dedicated backups, to support reporting, or to act as a cold standby.

Adjust Priority for Replica Set Member **(page 431)** Change the precedence given to a replica set members in an election for primary.

Prevent Secondary from Becoming Primary **(page 432)** Make a secondary member ineligible for election as primary.

Configure a Hidden Replica Set Member **(page 433)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

Configure a Delayed Replica Set Member **(page 434)** Configure a secondary member to keep a delayed copy of the data set in order to provide a rolling backup.

Configure Non-Voting Replica Set Member **(page 435)** Create a secondary member that keeps a copy of the data set but does not vote in an election.

Convert a Secondary to an Arbiter **(page 436)** Convert a secondary to an arbiter.

29.2.1 Adjust Priority for Replica Set Member

To change the value of the priority (page 475) in the replica set configuration, use the following sequence of commands in the mongo (page 1066) 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() (page 1015) to set the local variable cfg to the contents of the current replica set configuration, which is a *document*. The next three operations change the priority (page 475) value in the cfg document for the first three members configured in the members (page 474) array. The final operation calls rs.reconfig() (page 1016) with the argument of cfg to initialize the new configuration.

When updating the replica configuration object, access the replica set members in the members (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the _id (page 474) field in each document in the members (page 474) array.

If a member has priority (page 475) set to 0, it is ineligible to become *primary* and will not seek election. *Hidden members* (page 379), *delayed members* (page 379), and *arbiters* (page **??**) all have priority (page 475) set to 0.

All members have a priority (page 475) equal to 1 by default.

The value of `priority` (page 475) can be any floating point (i.e. decimal) number between `0` and `1000`. Priorities are only used to determine the preference in election. The priority value is used only in relation to other members. With the exception of members with a priority of `0`, the absolute value of the `priority` (page 475) value is irrelevant.

Replica sets will preferentially elect and maintain the primary status of the member with the highest `priority` (page 475) setting.

> **Warning:** Replica set reconfiguration can force the current primary to step down, leading to an election for primary in the replica set. Elections cause the current primary to close all open *client* connections.
> Perform routine replica set reconfiguration during scheduled maintenance windows.

See also:

The *Replica Reconfiguration Usage* (page 477) example revolves around changing the priorities of the `members` (page 474) of a replica set.

29.2.2 Prevent Secondary from Becoming Primary

To prevent a *secondary* member from ever becoming a *primary* in a *failover*, assign the secondary a priority of `0`, as described here. You can set this "secondary-only mode" for any member of the *replica set*, except the current primary. For a detailed description of secondary-only members and their purposes, see *Priority 0 Replica Set Members* (page 378).

To configure a member as secondary-only, set its `priority` (page 475) value to `0` in the `members` (page 474) document in its replica set configuration. Any member with a `priority` (page 475) equal to `0` will never seek *election* (page 389) and cannot become primary in any situation.

```
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "priority" : 0
}
```

MongoDB does not permit the current *primary* to have a priority of `0`. To prevent the current primary from again becoming a primary, you must first step down the current primary using `rs.stepDown()` (page 1018), and then you must *reconfigure the replica set* (page 477) with `rs.conf()` (page 1015) and `rs.reconfig()` (page 1016).

Example

As an example of modifying member priorities, assume a four-member replica set. Use the following sequence of operations to modify member priorities in the `mongo` (page 1066) shell connected to the primary. Identify each member by its array index in the `members` (page 474) array:

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

The sequence of operations reconfigures the set with the following priority settings:

- Member at `0` has a priority of `2` so that it becomes primary under most circumstances.
- Member at `1` has a priority of `1`, which is the default value. Member `1` becomes primary if no member with a *higher* priority is eligible.

- Member at 2 has a priority of 0.5, which makes it less likely to become primary than other members but doesn't prohibit the possibility.
- Member at 3 has a priority of 0. Member at 3 **cannot** become the *primary* member under any circumstances.

When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

> **Warning:**
> - The `rs.reconfig()` (page 1016) shell method can force the current primary to step down, which causes an *election* (page 389). When the primary steps down, the `mongod` (page 1049) closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
> - To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 425) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `priority` (page 475)
- *Adjust Priority for Replica Set Member* (page 431)
- *Replica Set Reconfiguration* (page 477)
- *Replica Set Elections* (page 389)

29.2.3 Configure a Hidden Replica Set Member

Hidden members are part of a *replica set* but cannot become *primary* and are invisible to client applications. Hidden members do, however, vote in *elections* (page 389). For a detailed description of hidden members and their purposes, see *Hidden Replica Set Members* (page 379).

If the `chainingAllowed` (page 476) setting allows secondary members to sync from other secondaries, MongoDB by default prefers non-hidden members over hidden members when selecting a sync target. MongoDB will only choose hidden members as a last resort. If you want a secondary to sync from a hidden member, use the `replSetSyncFrom` (page 869) database command to override the default sync target. See the documentation for `replSetSyncFrom` (page 869) before using the command.

See also:

Manage Chained Replication (page 450)

To configure a secondary member as hidden, set its `priority` (page 475) value to 0 and set its `hidden` (page 475) value to `true` in its member configuration:

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

Example

The following example hides the secondary member currently at the index `0` in the `members` (page 474) array. To configure a *hidden member*, use the following sequence of operations in a `mongo` (page 1066) shell connected to the primary, specifying the member to configure by its array index in the `members` (page 474) array:

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

After re-configuring the set, this secondary member has a priority of `0` so that it cannot become primary and is hidden. The other members in the set will not advertise the hidden member in the `isMaster` (page 871) or `db.isMaster()` (page 1008) output.

When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

Warning:

- The `rs.reconfig()` (page 1016) shell method can force the current primary to step down, which causes an *election* (page 389). When the primary steps down, the `mongod` (page 1049) closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 425) to ensure that members can quickly obtain a majority of votes in an election for primary.

Changed in version 2.0: For *sharded clusters* running with replica sets before 2.0, if you reconfigured a member as hidden, you *had* to restart `mongos` (page 1061) to prevent queries from reaching the hidden member.

Related Documents

- *Replica Set Reconfiguration* (page 477)
- *Replica Set Elections* (page 389)
- *Read Preference* (page 399)

29.2.4 Configure a Delayed Replica Set Member

To configure a delayed secondary member, set its `priority` (page 475) value to `0`, its `hidden` (page 475) value to `true`, and its `slaveDelay` (page 476) value to the number of seconds to delay.

Important: The length of the secondary `slaveDelay` (page 476) must fit within the window of the oplog. If the oplog is shorter than the `slaveDelay` (page 476) window, the delayed member cannot successfully replicate operations.

When you configure a delayed member, the delay applies both to replication and to the member's *oplog*. For details on delayed members and their uses, see *Delayed Replica Set Members* (page 379).

Example

The following example sets a 1-hour delay on a secondary member currently at the index `0` in the `members` (page 474) array. To set the delay, issue the following sequence of operations in a `mongo` (page 1066) shell connected to the primary:

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

After the replica set reconfigures, the delayed secondary member cannot become *primary* and is hidden from applications. The `slaveDelay` (page 476) value delays both replication and the member's *oplog* by 3600 seconds (1 hour).

When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

Warning:

- The `rs.reconfig()` (page 1016) shell method can force the current primary to step down, which causes an *election* (page 389). When the primary steps down, the `mongod` (page 1049) closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 425) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `slaveDelay` (page 476)
- *Replica Set Reconfiguration* (page 477)
- *Oplog Size* (page 405)
- *Change the Size of the Oplog* (page 439) tutorial
- *Replica Set Elections* (page 389)

29.2.5 Configure Non-Voting Replica Set Member

Non-voting members allow you to add additional members for read distribution beyond the maximum seven voting members. To configure a member as non-voting, set its `votes` (page 476) value to `0`.

Example

To disable the ability to vote in elections for the fourth, fifth, and sixth replica set members, use the following command sequence in the `mongo` (page 1066) shell connected to the primary. You identify each replica set member by its array index in the `members` (page 474) array:

```
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

This sequence gives `0` votes to the fourth, fifth, and sixth members of the set according to the order of the `members` (page 474) array in the output of `rs.conf()` (page 1015). This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

> **Warning:**
> - The `rs.reconfig()` (page 1016) shell method can force the current primary to step down, which causes an *election* (page 389). When the primary steps down, the `mongod` (page 1049) closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
> - To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 425) to ensure that members can quickly obtain a majority of votes in an election for primary.

In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary. Use `priority` (page 475) to control which members are more likely to become primary.

Related Documents

- `votes` (page 476)
- *Replica Set Reconfiguration* (page 477)
- *Replica Set Elections* (page 389)

29.2.6 Convert a Secondary to an Arbiter

If you have a *secondary* in a *replica set* that no longer needs to hold data but that needs to remain in the set to ensure that the set can *elect a primary* (page 389), you may convert the secondary to an *arbiter* (page **??**) using either procedure in this tutorial. Both procedures are operationally equivalent:

- You may operate the arbiter on the same port as the former secondary. In this procedure, you must shut down the secondary and remove its data before restarting and reconfiguring it as an arbiter.

  For this procedure, see *Convert Secondary to Arbiter and Reuse the Port Number* (page 436).

- Run the arbiter on a new port. In this procedure, you can reconfigure the server as an arbiter before shutting down the instance running as a secondary.

  For this procedure, see *Convert Secondary to Arbiter Running on a New Port Number* (page 437).

Convert Secondary to Arbiter and Reuse the Port Number

1. If your application is connecting directly to the secondary, modify the application so that MongoDB queries don't reach the secondary.

2. Shut down the secondary.

3. Remove the *secondary* from the *replica set* by calling the `rs.remove()` (page 1017) method. Perform this operation while connected to the current *primary* in the `mongo` (page 1066) shell:

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

4. Verify that the replica set no longer includes the secondary by calling the `rs.conf()` (page 1015) method in the `mongo` (page 1066) 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` (page 1049) instance. You can reuse the previous name. For example:

   ```
   mkdir /data/db
   ```

7. Restart the `mongod` (page 1049) 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` (page 1066) shell convert the secondary to an arbiter using the `rs.addArb()` (page 1015) method:

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

9. Verify the arbiter belongs to the replica set by calling the `rs.conf()` (page 1015) method in the `mongo` (page 1066) shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

Convert Secondary to Arbiter Running on a New Port Number

1. If your application is connecting directly to the secondary or has a connection string referencing the secondary, modify the application so that MongoDB queries don't reach the secondary.

2. Create a new, empty data directory to be used with the new port number. For example:

   ```
   mkdir /data/db-temp
   ```

3. Start a new `mongod` (page 1049) 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` (page 1066) shell connected to the current primary, convert the new `mongod` (page 1049) instance to an arbiter using the `rs.addArb()` (page 1015) method:

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

5. Verify the arbiter has been added to the replica set by calling the `rs.conf()` (page 1015) method in the `mongo` (page 1066) 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()` (page 1017) method in the `mongo` (page 1066) shell:

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

8. Verify that the replica set no longer includes the old secondary by calling the `rs.conf()` (page 1015) method in the `mongo` (page 1066) 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.

29.3 Replica Set Maintenance Tutorials

The following tutorials provide information in maintaining existing replica sets.

***Change the Size of the Oplog* (page 439)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

***Force a Member to Become Primary* (page 441)** Force a replica set member to become primary.

***Resync a Member of a Replica Set* (page 443)** Resync the data on a secondary that has fallen too far behind to catch up by way of normal replication.

***Configure Replica Set Tag Sets* (page 444)** Assign tags to replica set members for use in targeting read and write operations to specific members.

***Reconfigure a Replica Set with Unavailable Members* (page 448)** Reconfigure a replica set when a majority of replica set members are down or unreachable.

***Manage Chained Replication* (page 450)** Disable or enable chained replication. Chained replication occurs when a secondary replicates from another secondary instead of the primary.

***Change Hostnames in a Replica Set* (page 451)** Update the replica set configuration to reflect changes in members' hostnames.

***Configure a Secondary's Sync Target* (page 455)** Specify the member that a secondary member synchronizes from.

29.3.1 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 405) 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 Size* (page 405) topic in the *Replication* (page 367) document. For details on the how oplog size affects *delayed members* and affects *replication lag*, see the *Delayed Replica Set Members* (page 379) topic and *Check the Replication Lag* (page 455).

Overview

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

1. Shut down one of the *secondary* members of your *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 all secondaries that *could become* primary.
9. Step down the current *primary* with `rs.stepDown()` (page 1018), and repeat oplog resizing process above for the former primary (e.g. steps 2 through 7).

Procedure

The examples in this procedure use the following configuration:

- The active *replica set* is `rs0`.
- The replica set is running on port `27017`.
- The replica set is running with a `data directory` (page 1118) of `http://docs.mongodb.org/manualsrv/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` (page 1049) 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 1124) 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
```

Note: You can restore the backup using the `mongorestore` (page 1079) utility.

Connect to the instance using the `mongo` (page 1066) shell:

```
mongo --port 37017
```

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

   (a) In the `mongo` (page 1066) shell, enter the following command to use the `local` database to interact with the oplog:

   ```
   use local
   ```

   (b) Use the `db.collection.save()` (page 972) 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` (page 885) 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()`

(page 1018) method. This will force an election for primary. If the server's `priority` (page 475) 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.

29.3.2 Force a Member to Become Primary

Synopsis

You can force a *replica set* member to become *primary* by giving it a higher `priority` (page 475) value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `priority` (page 475) value to `0`, which means the member can never seek *election* (page 389) as primary. For more information, see *Priority 0 Replica Set Members* (page 378).

Procedures

Force a Member to be Primary by Setting its Priority High

Changed in version 2.0.

For more information on priorities, see `priority` (page 475).

This procedure assumes your current *primary* is `m1.example.net` and that you'd like to instead make `m3.example.net` primary. The procedure also assumes you have a three-member *replica set* with the configuration below. For more information on configurations, see *Replica Set Configuration Use* (page 477).

This procedure assumes this configuration:

```
{
    "_id" : "rs",
    "version" : 7,
    "members" : [
        {
            "_id" : 0,
            "host" : "m1.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "m2.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "m3.example.net:27017"
        }
    ]
}
```

1. In the `mongo` (page 1066) shell, use the following sequence of operations to make `m3.example.net` the primary:

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

This sets m3.example.net to have a higher local.system.replset.members[n].priority (page 475) value than the other mongod (page 1049) 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 475) 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() (page 1015) provides a wrapper around the replSetFreeze (page 865) 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 (page 1066) shell, run rs.status() (page 1017) to ensure your replica set is running as expected.
2. In a mongo (page 1066) shell connected to the mongod (page 1049) 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 (page 1066) shell connected the mongod (page 1049) running on mdb0.example.net, step down this instance that the mongod (page 1049) 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()` (page 1015) and `rs.stepDown()` (page 1018) methods that wrap the `replSetFreeze` (page 865) and `replSetStepDown` (page 868) commands.

29.3.3 Resync a Member of a Replica Set

When a secondary's replication process falls so far behind that *primary* overwrites oplog entries that the secondary has not yet replicated, that secondary cannot catch up and becomes "stale." When this occurs, you must completely resynchronize the member by removing its data and performing an initial sync.

To do so, use one of the following approaches:

- Restart the `mongod` (page 1049) with an empty data directory and let MongoDB's normal initial syncing feature restore the data. This is the more simple option, but may take longer to replace the data.

  See *Automatically Resync a Stale Member* (page 443).

- 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 All Datafiles from Another Member* (page 443).

Automatically Resync a Stale Member

This procedure relies on MongoDB's regular process for initial sync. This will restore the data on the stale member to reflect the current state of the set. For an overview of MongoDB initial sync process, see the *Replication Processes* (page 405) section.

To resync the stale member:

1. Stop the stale member's `mongod` (page 1049) instance. On Linux systems you can use *`mongod --shutdown`* Set *`--dbpath`* 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. By removing the data `dbpath` (page 1118), MongoDB will perform a complete resync. Consider making a backup first.

3. Restart the `mongod` (page 1049) instance on the member. For example:

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

   At this point, the `mongod` (page 1049) will perform an initial sync. The length of the initial sync may process depends on the size of the database and network connection between members of the replica set.

   Initial sync operations can impact the other members of the set and create additional traffic to the primary, and can only occur if another member of the set is accessible and up to date.

Resync by Copying All Datafiles from Another Member

This approach uses a copy of the data files from an existing member of the replica set, or a backup of the data files to "seed" the stale member.

The copy or backup of the data files **must** be sufficiently recent to allow the new member to catch up with the *oplog*, otherwise the member would need to perform an initial sync.

Note: In most cases you cannot copy data files from a running mongod (page 1049) instance to another, because the data files will change during the file copy operation. Consider the *Backup Strategies for MongoDB Systems* (page 133) documentation for several methods that you can use to capture a consistent snapshot of a running mongod (page 1049) instance.

Important: You must always copy the content of the local database when using data files to resync a member of a replica set.

After you have copied the data files from the "seed" source, start the mongod (page 1049) instance and allow it to apply all operations from the oplog until it reflects the current state of the replica set.

29.3.4 Configure Replica Set Tag Sets

Tag sets let you customize *write concern* and *read preferences* for a *replica set*. MongoDB stores tag sets in the replica set configuration object, which is the document returned by rs.conf() (page 1015), in the members[n].tags (page 475) sub-document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see *Replica Set Write Concern* (page 396) and *Tag Sets* (page 401).

Differences Between Read Preferences and Write Concerns

Custom read preferences and write concerns evaluate tags sets in different ways:

- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns do not use the value of a tag to select a member except to consider whether or not the value is unique.

For example, a tag set for a read operation may resemble the following document:

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

To fulfill such a read operation, a member would need to have both of these tags. Any of the following tag sets would satisfy this requirement:

```
{ "disk": "ssd", "use": "reporting" }
{ "disk": "ssd", "use": "reporting", "rack": "a" }
{ "disk": "ssd", "use": "reporting", "rack": "d" }
{ "disk": "ssd", "use": "reporting", "mem": "r"}
```

The following tag sets would *not* be able to fulfill this query:

```
{ "disk": "ssd" }
{ "use": "reporting" }
{ "disk": "ssd", "use": "production" }
{ "disk": "ssd", "use": "production", "rack": "k" }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Add Tag Sets to a Replica Set

Given the following replica set configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
             {
                     "_id" : 0,
                     "host" : "mongodb0.example.net:27017"
             },
             {
                     "_id" : 1,
                     "host" : "mongodb1.example.net:27017"
             },
             {
                     "_id" : 2,
                     "host" : "mongodb2.example.net:27017"
             }
     ]
}
```

You could add tag sets to the members of this replica set with the following command sequence in the `mongo` (page 1066) 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()` (page 1015) would resemble the following:

```
{
    "_id" : "rs0",
    "version" : 2,
    "members" : [
             {
                     "_id" : 0,
                     "host" : "mongodb0.example.net:27017",
                     "tags" : {
                             "dc": "east",
                             "use": "production"
                     }
             },
             {
                     "_id" : 1,
                     "host" : "mongodb1.example.net:27017",
                     "tags" : {
                             "dc": "east",
                             "use": "reporting"
                     }
             },
             {
                     "_id" : 2,
                     "host" : "mongodb2.example.net:27017",
                     "tags" : {
                             "use": "production"
```

```
                    }
            }
    ]
}
```

Important: In tag sets, all tag values must be strings.

Custom Multi-Datacenter Write Concerns

Given a five member replica set with members in two data centers:

1. a facility `VA` tagged `dc.va`
2. a facility `GTO` tagged `dc.gto`

Create a custom write concern to require confirmation from two data centers using replica set tags, using the following sequence of operations in the `mongo` (page 1066) shell:

1. Create a replica set configuration JavaScript object `conf`:

   ```
   conf = rs.conf()
   ```

2. Add tags to the replica set members reflecting their locations:

   ```
   conf.members[0].tags = { "dc.va": "rack1"}
   conf.members[1].tags = { "dc.va": "rack2"}
   conf.members[2].tags = { "dc.gto": "rack1"}
   conf.members[3].tags = { "dc.gto": "rack2"}
   conf.members[4].tags = { "dc.va": "rack1"}
   rs.reconfig(conf)
   ```

3. Create a custom `getLastErrorModes` (page 477) setting to ensure that the write operation will propagate to at least one member of each facility:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 1, "dc.gto": 1}}
   ```

4. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

To ensure that a write operation propagates to at least one member of the set in both data centers, use the `MultipleDC` write concern mode as follows:

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

Alternatively, if you want to ensure that each write operation propagates to at least 2 racks in each facility, reconfigure the replica set as follows in the `mongo` (page 1066) shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 477) value to require two different values of both `dc.va` and `dc.gto`:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 2, "dc.gto": 2}}
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

```
rs.reconfig(conf)
```

Now, the following write concern operation will only return after the write operation propagates to at least two different racks in the each facility:

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

Configure Tag Sets for Functional Segregation of Read and Write Operations

Given a replica set with tag sets that reflect:

- data center facility,
- physical rack location of instance, and
- storage system (i.e. disk) type.

Where each member of the set has a tag set that resembles one of the following: [3]

```
{"dc.va": "rack1", disk:"ssd", ssd: "installed" }
{"dc.va": "rack2", disk:"raid"}
{"dc.gto": "rack1", disk:"ssd", ssd: "installed" }
{"dc.gto": "rack2", disk:"raid"}
{"dc.va": "rack1", disk:"ssd", ssd: "installed" }
```

To target a read operation to a member of the replica set with a disk type of `ssd`, you could use the following tag set:

```
{ disk: "ssd" }
```

However, to create comparable write concern modes, you would specify a different set of `getLastErrorModes` (page 477) configuration. Consider the following sequence of operations in the `mongo` (page 1066) shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 477) value to configure two write concern modes:

   ```
   conf.settings = {
                    "getLastErrorModes" : {
                            "ssd" : {
                                       "ssd" : 1
                            },
                            "MultipleDC" : {
                                    "dc.va" : 1,
                                   "dc.gto" : 1
                            }
                    }
                  }
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now you can specify the `MultipleDC` write concern mode, as in the following operation, to ensure that a write operation propagates to each data center.

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

[3] Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.

Additionally, you can specify the `ssd` write concern mode to ensure that a write operation propagates to at least one instance with an SSD.

29.3.5 Reconfigure a Replica Set with Unavailable Members

To reconfigure a *replica set* when a **minority** of members are unavailable, use the `rs.reconfig()` (page 1016) operation on the current *primary*, following the example in the *Replica Set Reconfiguration Procedure* (page 477).

This document provides the following options for re-configuring a replica set when a **majority** of members are *not* accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 448)
- *Reconfigure by Replacing the Replica Set* (page 449)

You may need to use one of these procedures, for example, in a geographically distributed replica set, where *no* local group of members can reach a majority. See *Replica Set Elections* (page 389) for more information on this situation.

Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of *replica set* members are down or unreachable. You connect to any surviving member and use the `force` option to the `rs.reconfig()` (page 1016) method.

The `force` option forces a new configuration onto the. Use this procedure only to recover from catastrophic interruptions. Do not use `force` every time you reconfigure. Also, do not use the `force` option in any automatic scripts and do not use `force` when there is still a *primary*.

To force reconfiguration:

1. Back up a surviving member.
2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

   ```
   cfg = rs.conf()

   printjson(cfg)
   ```

3. On the same member, remove the down and unreachable members of the replica set from the `members` (page 474) array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

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

4. On the same member, reconfigure the set by using the `rs.reconfig()` (page 1016) command with the `force` option set to `true`:

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

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

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

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

Reconfigure by Replacing the Replica Set

Use the following procedure **only** for versions of MongoDB prior to version 2.0. If you're running MongoDB 2.0 or later, use the above procedure, *Reconfigure by Forcing the Reconfiguration* (page 448).

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()` (page 1016) command according to the examples in *Example Reconfiguration Operations* (page 477).

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

Reconfigure by Turning Off Replication

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

1. Stop the surviving `mongod` (page 1049) 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` to the data directory of your `mongod` (page 1049) instance.

2. Create a backup of the data directory (i.e. `dbpath` (page 1118)) 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` (page 1049) instances *without* the `--replSet` parameter.

   The data is now accessible and provided by a single server that is not a replica set member. Clients can use this server for both reads and writes.

When possible, re-deploy a replica set to provide redundancy and to protect your deployment from operational interruption.

Reconfigure by "Breaking the Mirror"

This option selects a surviving *replica set* member to be the new *primary* and to "seed" a new replica set. In the following procedure, the new primary is `db0.example.net`. MongoDB copies the data from `db0.example.net` to all the other members.

1. Stop the surviving `mongod` (page 1049) 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` to the data directory of your `mongod` (page 1049) instance.

2. Move the data directories (i.e. `dbpath` (page 1118)) 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` (page 1066) shell and run `rs.initiate()` (page 1016) to initiate the replica set.

6. Add the other set members using `rs.add()` (page 1014). For example, to add a member running on `db1.example.net` at port `27017`, issue the following command:

```
rs.add("db1.example.net:27017")
```

MongoDB performs an initial sync on the added members by copying all data from `db0.example.net` to the added members.

29.3.6 Manage Chained Replication

Starting in version 2.0, MongoDB supports chained replication. A chained replication occurs when a *secondary* member replicates from another secondary member instead of from the *primary*. This might be the case, for example, if a secondary selects its replication target based on ping time and if the closest member is another secondary.

Chained replication can reduce load on the primary. But chained replication can also result in increased replication lag, depending on the topology of the network.

New in version 2.2.2.

You can use the `chainingAllowed` (page 476) setting in *Replica Set Configuration* (page 473) to disable chained replication for situations where chained replication is causing lag.

MongoDB enables chained replication by default. This procedure describes how to disable it and how to re-enable it.

Note: If chained replication is disabled, you still can use `replSetSyncFrom` (page 869) to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Disable Chained Replication

To disable chained replication, set the `chainingAllowed` (page 476) field in *Replica Set Configuration* (page 473) to `false`.

You can use the following sequence of commands to set `chainingAllowed` (page 476) to `false`:

1. Copy the configuration settings into the `cfg` object:

```
cfg = rs.config()
```

2. Take note of whether the current configuration settings contain the `settings` sub-document. If they do, skip this step.

Warning: To avoid data loss, skip this step if the configuration settings contain the `settings` sub-document.

If the current configuration settings **do not** contain the `settings` sub-document, create the sub-document by issuing the following command:

```
cfg.settings = { }
```

3. Issue the following sequence of commands to set `chainingAllowed` (page 476) to `false`:

```
cfg.settings.chainingAllowed = false
rs.reconfig(cfg)
```

Re-enable Chained Replication

To re-enable chained replication, set `chainingAllowed` (page 476) to `true`. You can use the following sequence of commands:

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```

29.3.7 Change Hostnames in a Replica Set

For most *replica sets*, the hostnames in the `host` (page 474) field never change. However, if organizational needs change, you might need to migrate some or all host names.

Note: Always use resolvable hostnames for the value of the `host` (page 474) field in the replica set configuration to avoid confusion and complexity.

Overview

This document provides two separate procedures for changing the hostnames in the `host` (page 474) field. Use either of the following approaches:

- *Change hostnames without disrupting availability* (page 452). This approach ensures your applications will always be able to read and write data to the replica set, but the approach can take a long time and may incur downtime at the application layer.

  If you use the first procedure, you must configure your applications to connect to the replica set at both the old and new locations, which often requires a restart and reconfiguration at the application layer and which may affect the availability of your applications. Re-configuring applications is beyond the scope of this document.

- *Stop all members running on the old hostnames at once* (page 453). This approach has a shorter maintenance window, but the replica set will be unavailable during the operation.

See also:

Replica Set Reconfiguration Process (page 477), *Deploy a Replica Set* (page 416), and *Add Members to a Replica Set* (page 427).

Assumptions

Given a *replica set* with three members:

- `database0.example.com:27017` (the *primary*)
- `database1.example.com:27017`
- `database2.example.com:27017`

And with the following `rs.conf()` (page 1015) output:

```
{
    "_id" : "rs",
    "version" : 3,
    "members" : [
        {
            "_id" : 0,
            "host" : "database0.example.com:27017"
        },
        {
            "_id" : 1,
            "host" : "database1.example.com:27017"
        },
        {
            "_id" : 2,
            "host" : "database2.example.com:27017"
        }
    ]
}
```

The following procedures change the members' hostnames as follows:

- `mongodb0.example.net:27017` (the primary)
- `mongodb1.example.net:27017`
- `mongodb2.example.net:27017`

Use the most appropriate procedure for your deployment.

Change Hostnames while Maintaining Replica Set Availability

This procedure uses the above *assumptions* (page 452).

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` (page 1066) 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 `host` (page 474) value where `n` is `1`:

   ```
   cfg = rs.conf()

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

   rs.reconfig(cfg)
   ```

See *Replica Set Configuration* (page 473) 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` (page 1066) shell connected to the primary and step down the primary using `replSetStepDown` (page 868). In the `mongo` (page 1066) shell, use the `rs.stepDown()` (page 1018) 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` (page 1066) shell and reconfigure the `host` (page 474) 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` (page 1066) shell connected to the primary.

7. To confirm the new configuration, call `rs.conf()` (page 1015) in the `mongo` (page 1066) shell.

Your output should resemble:

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

Change All Hostnames at the Same Time

This procedure uses the above *assumptions* (page 452).

1. Stop all members in the *replica set*.

2. Restart each member *on a different port* and *without* using the *--replSet* run-time option. Changing the port number during maintenance prevents clients from connecting to this host while

you perform maintenance. Use the member's usual *--dbpath*, which in this example is `http://docs.mongodb.org/manualdata/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` (page 1066) shell connected to the `mongod` (page 1049) 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` (page 1049) process on the member.

4. After re-configuring all members of the set, start each `mongod` (page 1049) instance in the normal way: use the usual port number and use the *--replSet* option. For example:

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

5. Connect to one of the `mongod` (page 1049) instances using the `mongo` (page 1066) shell. For example:

   ```
   mongo --port 27017
   ```

6. To confirm the new configuration, call `rs.conf()` (page 1015) in the `mongo` (page 1066) 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"
           }
   ```

```
    ]
}
```

29.3.8 Configure a Secondary's Sync Target

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target for pulling *oplog* entries temporarily. The following operations provide access to this functionality:

- `replSetSyncFrom` (page 869) command, or
- `rs.syncFrom()` (page 1018) helper in the `mongo` (page 1066) shell

Only modify the default sync logic as needed, and always exercise caution. `rs.syncFrom()` (page 1018) will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run `rs.syncFrom()` (page 1018) operation *before* initial sync.

If you run `rs.syncFrom()` (page 1018) during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Note: `replSetSyncFrom` (page 869) and `rs.syncFrom()` (page 1018) provide a temporary override of default behavior. If:

- the `mongod` (page 1049) instance restarts,
- the connection to the sync target closes, or
- Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set;

then, the `mongod` (page 1049) instance will revert to the default sync logic and target.

29.4 Troubleshoot Replica Sets

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

29.4.1 Check Replica Set Status

To display the current state of the replica set and current state of each member, run the `rs.status()` (page 1017) method in a `mongo` (page 1066) shell connected to the replica set's *primary*. For descriptions of the information displayed by `rs.status()` (page 1017), see *replSetGetStatus* (page 865).

Note: The `rs.status()` (page 1017) method is a wrapper that runs the `replSetGetStatus` (page 865) database command.

29.4.2 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` (page 1066) shell connected to the primary, call the `db.printSlaveReplicationInfo()` (page 1011) 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()` (page 1017) method is a wrapper around the `replSetGetStatus` (page 865) database command.

- Monitor the rate of replication by watching the oplog time in the "replica" graph in the MongoDB Management 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. For best results, configure *write concern* (page 395) to require confirmation of replication to secondaries, as described in *replica set write concern* (page 396). This prevents write operations from returning if replication cannot keep up with the write load.

  Use the *database profiler* to see if there are slow queries or long-running operations that correspond to the incidences of lag.

- **Appropriate Write Concern**

  If you are performing a large data ingestion or bulk load operation that requires a large number of writes to the primary, particularly with *unacknowledged write concern* (page 395), the secondaries will not be able to read the oplog fast enough to keep up with changes.

  To prevent this, require *write acknowledgment or journaled write concern* (page 54) after every 100, 1,000, or an another interval to provide an opportunity for secondaries to catch up with the primary.

  For more information see:

  – *Replica Acknowledge Write Concern* (page 396)

  – *Replica Set Write Concern* (page 57)

  – *Oplog Size* (page 405)

29.4.3 Test Connections Between all Members

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both "directions." Networking topologies and firewall configurations prevent normal and required connectivity, which can block replication.

Consider the following example of a bidirectional test of networking:

Example

Given a replica set with three members running on three separate hosts:

- `m1.example.net`
- `m2.example.net`
- `m3.example.net`

1. Test the connection from `m1.example.net` to the other hosts with the following operation set `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.

29.4.4 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` (page 1066) shell and run the `db.printReplicationInfo()` (page 1010) method.

The output displays the size of the oplog and the date ranges of the operations contained in the oplog. In the following example, the oplog is about 10MB and is able to fit about 26 hours (94400 seconds) of operations:

```
configured oplog size:   10.10546875MB
log length start to end: 94400 (26.22hrs)
oplog first event time:  Mon Mar 19 2012 13:50:38 GMT-0400 (EDT)
oplog last event time:   Wed Oct 03 2012 14:59:10 GMT-0400 (EDT)
now:                     Wed Oct 03 2012 15:00:21 GMT-0400 (EDT)
```

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations; however, many users prefer to have 72 hours or even a week's work of operations.

For more information on how oplog size affects operations, see:

- *Oplog Size* (page 405),
- *Delayed Replica Set Members* (page 379), and
- *Check the Replication Lag* (page 455).

Note: You normally want the oplog to be the same size on all members. If you resize the oplog, resize it on all members.

To change oplog size, see the *Change the Size of the Oplog* (page 439) tutorial.

29.4.5 Oplog Entry Timestamp Error

Consider the following error in `mongod` (page 1049) output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection.  Terminating mongod after 30
<timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the `ts` field in the last *oplog* entry causes this error. The correct data type is Timestamp.

Check the type of the `ts` value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the `ts` value is a Timestamp. The `$type` (page 795) operator allows you to select *BSON type* 17, is the Timestamp data type.

If the queries don't return the same document, then the last document in the oplog has the wrong data type in the `ts` field.

Example

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where `ts` has the `Timestamp` type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the `ts` field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
                    { $set: { ts: new Timestamp(1347982456000, 1)}})
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

29.4.6 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` (page 1066) 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.

CHAPTER 30

Replication Reference

30.1 Replication Methods in the mongo Shell

Name	Description
`rs.add()` (page 1014)	Adds a member to a replica set.
`rs.addArb()` (page 1015)	Adds an *arbiter* to a replica set.
`rs.conf()` (page 1015)	Returns the replica set configuration document.
`rs.freeze()` (page 1015)	Prevents the current member from seeking election as primary for a period of time.
`rs.help()` (page 1016)	Returns basic help text for *replica set* functions.
`rs.initiate()` (page 1016)	Initializes a new replica set.
`rs.reconfig()` (page 1016)	Re-configures a replica set by applying a new replica set configuration object.
`rs.remove()` (page 1017)	Remove a member from a replica set.
`rs.slaveOk()` (page 1017)	Sets the `slaveOk` property for the current connection. Deprecated. Use `readPref()` (page 989) and `Mongo.setReadPref()` (page 1035) to set *read preference*.
`rs.status()` (page 1017)	Returns a document with information about the state of the replica set.
`rs.stepDown()` (page 1018)	Causes the current *primary* to become a secondary which forces an *election*.
`rs.syncFrom()` (page 1018)	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

30.2 Replication Database Commands

Name	Description
replSetFreeze (page 865)	Prevents the current member from seeking election as *primary* for a period of time.
replSetGetStatus (page 865)	Returns a document that reports on the status of the replica set.
replSetInitiate (page 866)	Initializes a new replica set.
replSetMaintenance (page 867)	Enables or disables a maintenance mode, which puts a *secondary* node in a RECOVERING state.
replSetReconfig (page 868)	Applies a new configuration to an existing replica set.
replSetStepDown (page 868)	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
replSetSyncFrom (page 869)	Explicitly override the default logic for selecting a member to replicate from.
resync (page 870)	Forces a mongod (page 1049) to re-synchronize from the *master*. For master-slave replication only.
applyOps (page 870)	Internal command that applies *oplog* entries to the current data set.
isMaster (page 871)	Displays information about this member's role in the replica set, including whether it is the master.
getoptime (page 872)	Internal command to support replication, returns the optime.

30.3 Replica Set Reference Documentation

***Replica Set Commands* (page 462)** A quick reference for all *commands* and `mongo` (page 1066) shell methods that support replication.

***Replica Set Configuration* (page 473)** Complete documentation of the *replica set* configuration object returned by `rs.conf()` (page 1015).

***The local Database* (page 478)** Complete documentation of the content of the `local` database that `mongod` (page 1049) instances use to support replication.

***Replica Set Member States* (page 480)** Reference for the replica set member states.

30.3.1 Replica Set Commands

This reference collects documentation for all *JavaScript methods* (page 462) for the `mongo` (page 1066) shell that support *replica set* functionality, as well as all *database commands* (page 466) related to replication function.

See *Replication* (page 367), for a list of all replica set documentation.

JavaScript Methods

The following methods apply to replica sets. For a complete list of all methods, see *mongo Shell Methods* (page 944).

`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` (page 865) *database command*.

`db.isMaster()`

Returns A document that describes the role of the `mongod` (page 1049) instance.

If the `mongod` (page 1049) is a member of a *replica set*, then the `ismaster` (page 871) and `secondary` (page 872) fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

`isMaster` (page 871) for the complete documentation of the output of `isMaster()` (page 1008).

Description

`rs.initiate(configuration)`

Initiates a *replica set*. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set.

The `rs.initiate()` (page 1016) method has the following parameter:

param document configuration A *document* that specifies *configuration settings* (page 473) for tne new replica set. If a configuration is not specified, MongoDB uses a default configuration.

The `rs.initiate()` (page 1016) method provides a wrapper around the "`replSetInitiate` (page 866)" *database command*.

Replica Set Configuration

See *Member Configuration Tutorials* (page 431) and *Replica Set Configuration* (page 473) for examples of replica set configuration and invitation objects.

`rs.conf()`

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

`rs.config()`

`rs.config()` (page 1015) is an alias of `rs.conf()` (page 1015).

Definition

`rs.reconfig(configuration, force)`

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

param document configuration A *document* that specifies the configuration of a replica set.

param document force "If set as `{ force: true }`, this forces the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to term:*rollback* situations."

`rs.reconfig()` (page 1016) overwrites the existing replica set configuration. Retrieve the current configuration object with `rs.conf()` (page 1015), modify the configuration as needed and then use `rs.reconfig()` (page 1016) to submit the modified configuration object.

`rs.reconfig()` (page 1016) provides a wrapper around the "`replSetReconfig` (page 868)" *database command*.

Examples

To reconfigure a replica set, use the following sequence of operations:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf)
```

If you want to force the reconfiguration if a majority of the set is not connected to the current member, or you are issuing the command against a secondary, use the following form:

```
conf = rs.conf()

// modify conf to change configuration

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

Warning: Forcing a `rs.reconfig()` (page 1016) can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

See also:

Replica Set Configuration (page 473) and *Replica Set Tutorials* (page 415).

Definition

`rs.`**`add`**(*host*, *arbiterOnly*)

Adds a member to a *replica set*.

param string,document host The new member to add to the replica set. If a string, specifies the hostname and optionally the port number for the new member. If a document, specifies a replica set members document, as found in the `members` (page 474) array. To view a replica set's members array, run `rs.conf()` (page 1015).

param boolean arbiterOnly Applies only if the `<host>` value is a string. If `true`, the added host is an arbiter."

You may specify new hosts in one of two ways:

1. as a "hostname" with an optional port number to use the default configuration as in the *Add a Member to an Existing Replica Set* (page 429) example.
2. as a configuration *document*, as in the *Configure and Add a Member* (page 429) example.

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

`rs.add()` (page 1014) provides a wrapper around some of the functionality of the "`replSetReconfig` (page 868)" *database command* and the corresponding shell helper `rs.reconfig()` (page 1016). See the *Replica Set Configuration* (page 473) document for full documentation of all replica set configuration options.

Example

To add a `mongod` (page 1049) accessible on the default port `27017` running on the host `mongodb3.example.net`, use the following `rs.add()` (page 1014) invocation:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

To add `mongodb3.example.net` as a *secondary-only* (page 378) member of set, use the following form of `rs.add()` (page 1014):

```
rs.add( { "_id": 3, "host": "mongodbd3.example.net:27017", "priority": 0 } )
```

Replace, `3` with the next unused `_id` value in the replica set. See `rs.conf()` (page 1015) to see the existing `_id` values in the replica set configuration document.

See the *Replica Set Configuration* (page 473) and *Replica Set Tutorials* (page 415) documents for more information.

Description

`rs.addArb(host)`

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

The `rs.addArb()` (page 1015) method takes the following parameter:

param string host Specifies the hostname and optionally the port number of the arbiter member to add to replica set.

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

Description

`rs.stepDown(seconds)`

Forces the current *replica set* member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current member is not the primary.

The `rs.stepDown()` (page 1018) method has the following parameter:

param number seconds The duration of time that the stepped-down member attempts to avoid re-election as primary. If this parameter is not specified, the method uses the default value of 60 seconds.

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

`rs.stepDown()` (page 1018) provides a wrapper around the *database command* `replSetStepDown` (page 868).

Description

`rs.freeze(seconds)`

Makes the current *replica set* member ineligible to become *primary* for the period specified.

The `rs.freeze()` (page 1015) method has the following parameter:

param number seconds The duration the member is ineligible to become primary.

`rs.freeze()` (page 1015) provides a wrapper around the *database command* `replSetFreeze` (page 865).

Definition

`rs.remove(hostname)`

Removes the member described by the `hostname` parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

The `rs.remove()` (page 1017) method has the following parameter:

param string hostname The hostname of a system in the replica set.

Note: Before running the `rs.remove()` (page 1017) 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()` (page 1017), but it remains good practice.

`rs.slaveOk()`

Provides a shorthand for the following operation:

```
db.getMongo().setSlaveOk()
```

This allows the current connection to allow read operations to run on *secondary* members. See the `readPref()` (page 989) method for more fine-grained control over *read preference* (page 398) in the `mongo` (page 1066) shell.

`db.isMaster()`

Returns A document that describes the role of the `mongod` (page 1049) instance.

If the `mongod` (page 1049) is a member of a *replica set*, then the `ismaster` (page 871) and `secondary` (page 872) fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

`isMaster` (page 871) for the complete documentation of the output of `isMaster()` (page 1008).

`rs.help()`

Returns a basic help text for all of the *replication* (page 367) related shell functions.

`rs.syncFrom()`

New in version 2.2.

Provides a wrapper around the `replSetSyncFrom` (page 869), which allows administrators to configure the member of a replica set that the current member will pull data from. Specify the name of the member you want to replicate from in the form of `[hostname]:[port]`.

See `replSetSyncFrom` (page 869) for more details.

Database Commands

The following commands apply to replica sets. For a complete list of all commands, see *Database Commands* (page 833).

Definition

isMaster

`isMaster` (page 871) returns a document that describes the role of the `mongod` (page 1049) instance.

If the instance is a member of a replica set, then `isMaster` (page 871) returns a subset of the replica set configuration and status including whether or not the instance is the *primary* of the replica set.

When sent to a `mongod` (page 1049) instance that is not a member of a replica set, `isMaster` (page 871) returns a subset of this information.

MongoDB *drivers* and *clients* use `isMaster` (page 871) to determine the state of the replica set members and to discover additional members of a *replica set*.

The `db.isMaster()` (page 1008) method in the `mongo` (page 1066) shell provides a wrapper around `isMaster` (page 871).

The command takes the following form:

```
{ isMaster: 1 }
```

See also:

`db.isMaster()` (page 1008)

Output

All Instances

The following `isMaster` (page 871) fields are common across all roles:

`isMaster.`**`ismaster`**

A boolean value that reports when this node is writable. If `true`, then this instance is a *primary* in a *replica set*, or a *master* in a master-slave configuration, or a `mongos` (page 1061) instance, or a standalone `mongod` (page 1049).

This field will be `false` if the instance is a *secondary* member of a replica set or if the member is an *arbiter* of a replica set.

`isMaster.`**`maxBsonObjectSize`**

The maximum permitted size of a *BSON* object in bytes for this `mongod` (page 1049) process. If not provided, clients should assume a max size of "`4 * 1024 * 1024`".

`isMaster.`**`maxMessageSizeBytes`**

New in version 2.4.

The maximum permitted size of a *BSON* wire protocol message. The default value is `48000000` bytes.

`isMaster.`**`localTime`**

New in version 2.2.

Returns the local server time in UTC. This value is an *ISO date*.

Sharded Instances

`mongos` (page 1061) instances add the following field to the `isMaster` (page 871) response document:

`isMaster.`**`msg`**

Contains the value `isdbgrid` when `isMaster` (page 871) returns from a `mongos` (page 1061) instance.

Replica Sets

isMaster (page 871) contains these fields when returned by a member of a replica set:

isMaster.**setName**
The name of the current :replica set.

isMaster.**secondary**
A boolean value that, when true, indicates if the mongod (page 1049) is a *secondary* member of a *replica set*.

isMaster.**hosts**
An array of strings in the format of "[hostname]:[port]" that lists all members of the *replica set* that are neither *hidden*, *passive*, nor *arbiters*.

Drivers use this array and the isMaster.passives (page 872) to determine which members to read from.

isMaster.**passives**
An array of strings in the format of "[hostname]:[port]" listing all members of the *replica set* which have a priority (page 475) of 0.

This field only appears if there is at least one member with a priority (page 475) of 0.

Drivers use this array and the isMaster.hosts (page 872) to determine which members to read from.

isMaster.**arbiters**
An array of strings in the format of "[hostname]:[port]" listing all members of the *replica set* that are *arbiters*.

This field only appears if there is at least one arbiter in the replica set.

isMaster.**primary**
A string in the format of "[hostname]:[port]" listing the current *primary* member of the replica set.

isMaster.**arbiterOnly**
A boolean value that , when true, indicates that the current instance is an *arbiter*. The arbiterOnly (page 872) field is only present, if the instance is an arbiter.

isMaster.**passive**
A boolean value that, when true, indicates that the current instance is *hidden*. The passive (page 872) field is only present for hidden members.

isMaster.**hidden**
A boolean value that, when true, indicates that the current instance is *hidden*. The hidden (page 872) field is only present for hidden members.

isMaster.**tags**
A document that lists any tags assigned to this member. This field is only present if there are tags assigned to the member. See *Configure Replica Set Tag Sets* (page 444) for more information.

isMaster.**me**
The [hostname]:[port] of the member that returned isMaster (page 871).

resync
The resync (page 870) command forces an out-of-date slave mongod (page 1049) instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does not apply to replica sets.

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

replSetFreeze
The replSetFreeze (page 865) command prevents a replica set member from seeking election for the speci-

fied number of seconds. Use this command in conjunction with the `replSetStepDown` (page 868) command to make a different node in the replica set a primary.

The `replSetFreeze` (page 865) 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` (page 1049) process also unfreezes a replica set member.

`replSetFreeze` (page 865) is an administrative command, and you must issue it against the *admin database*.

Definition

replSetGetStatus

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

The value specified does not affect the output of the command. Data provided by this command derives from data included in heartbeats sent to the current instance by other members of the replica set. Because of the frequency of heartbeats, these data can be several seconds out of date.

You can also access this functionality through the `rs.status()` (page 1017) helper in the `mongo` (page 1066) shell.

The `mongod` (page 1049) must have replication enabled and be a member of a replica set for the for `replSetGetStatus` (page 865) to return successfully.

Output

replSetGetStatus.**set**

The `set` value is the name of the replica set, configured in the `replSet` (page 1124) setting. This is the same value as `_id` (page 474) in `rs.conf()` (page 1015).

replSetGetStatus.**date**

The value of the `date` field is an *ISODate* of the current time, according to the current server. Compare this to the value of the `lastHeartbeat` (page 866) to find the operational lag between the current host and the other hosts in the set.

replSetGetStatus.**myState**

The value of `myState` (page 865) is an integer between `0` and `10` that represents the *replica state* (page 480) of the current member.

replSetGetStatus.**members**

The `members` field holds an array that contains a document for every member in the replica set.

replSetGetStatus.members.**name**

The `name` field holds the name of the server.

replSetGetStatus.members.**self**

The `self` field is only included in the document for the current `mongod` instance in the members array. It's value is `true`.

replSetGetStatus.members.**errmsg**
This field contains the most recent error or status message received from the member. This field may be empty (e.g. `""`) in some cases.

replSetGetStatus.members.**health**
The `health` value is only present for the other members of the replica set (i.e. not the member that returns `rs.status` (page 1017).) This field conveys if the member is up (i.e. `1`) or down (i.e. `0`.)

replSetGetStatus.members.**state**
The value of `state` (page 866) is an array of documents, each containing an integer between `0` and `10` that represents the *replica state* (page 480) of the corresponding member.

replSetGetStatus.members.**stateStr**
A string that describes `state` (page 866).

replSetGetStatus.members.**uptime**
The `uptime` (page 866) 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()` (page 1017) data.

replSetGetStatus.members.**optime**
A document that contains information regarding the last operation from the operation log that this member has applied.

replSetGetStatus.members.optime.**t**
A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

replSetGetStatus.members.optime.**i**
An incremented field, which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

replSetGetStatus.members.**optimeDate**
An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from `lastHeartbeat` (page 866) this member is either experiencing "replication lag" *or* there have not been any new operations since the last update. Compare `members.optimeDate` between all of the members of the set.

replSetGetStatus.members.**lastHeartbeat**
The `lastHeartbeat` value provides an *ISODate* formatted date of the last heartbeat received from this member. Compare this value to the value of the `date` (page 865) field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` (page 1017) data.

replSetGetStatus.members.**pingMS**
The `pingMS` represents the number of milliseconds (ms) that a round-trip packet takes to travel between the remote member and the local instance.

This value does not appear for the member that returns the `rs.status()` (page 1017) data.

replSetGetStatus.**syncingTo**
The `syncingTo` field is only present on the output of `rs.status()` (page 1017) on *secondary* and recovering members, and holds the hostname of the member from which this instance is syncing.

replSetInitiate
The `replSetInitiate` (page 866) 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() (page 1016) 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 473)," "*Replica Set Tutorials* (page 415)," and "*Replica Set Reconfiguration* (page 477)."

replSetMaintenance

The replSetMaintenance (page 867) 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 (page 867) 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.

Important: On secondaries, the compact (page 890) command forces the secondary to enter RECOVERING (page 482) state. This prevents clients from reading during compaction. Once the operation finishes, the secondary returns to SECONDARY (page 481) state.

See *Replica Set Member States* (page 480) for more information about replica set member states. Refer to the "partial script for automating step down and compaction" for an example of this procedure.

---

replSetReconfig

The replSetReconfig (page 868) 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()` (page 1016) method.

Be aware of the following `replSetReconfig` (page 868) 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` (page 868) 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` (page 868) 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` (page 868) obtains a special mutually exclusive lock to prevent more than one `replSetReconfig` (page 868) operation from occurring at the same time.

Description

`replSetSyncFrom`

New in version 2.2.

Explicitly configures which host the current `mongod` (page 1049) pulls *oplog* entries from. This operation is useful for testing different patterns and in situations where a set member is not replicating from the desired host.

The `replSetSyncFrom` (page 869) command has the following form:

```
{ replSetSyncFrom: "hostname<:port>" }
```

The `replSetSyncFrom` (page 869) command has the following field:

field string replSetSyncFrom The name and port number of the replica set member that this member should replicate from. Use the `[hostname]:[port]` form.

The Target Member

The member to replicate from must be a valid source for data in the set. The member cannot be:

- The same as the `mongod` (page 1049) on which you run `replSetSyncFrom` (page 869). In other words, a member cannot replicate from itself.

- An arbiter, because arbiters do not hold data.
- A member that does not build indexes.
- An unreachable member.
- A `mongod` (page 1049) instance that is not a member of the same replica set.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` (page 1049) will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` (page 869) during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Run from the `mongo` Shell

To run the command in the `mongo` (page 1066) shell, use the following invocation:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } )
```

You may also use the `rs.syncFrom()` (page 1018) helper in the `mongo` (page 1066) shell in an operation with the following form:

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

Note: `replSetSyncFrom` (page 869) and `rs.syncFrom()` (page 1018) provide a temporary override of default behavior. If:

- the `mongod` (page 1049) instance restarts,
- the connection to the sync target closes, or
- Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set;

then, the `mongod` (page 1049) instance will revert to the default sync logic and target.

30.3.2 Replica Set Configuration

Synopsis

This reference provides an overview of replica set configuration options and settings.

Use `rs.conf()` (page 1015) in the `mongo` (page 1066) shell to retrieve this configuration. Note that default values are not explicitly displayed.

Example Configuration Document

The following document provides a representation of a replica set configuration document. Angle brackets (e.g. < and >) enclose all optional fields.

```
{
  _id : <setname>,
  version: <int>,
  members: [
    {
```

```
      _id : <ordinal>,
      host : hostname<:port>,
      <arbiterOnly : <boolean>,>
      <buildIndexes : <boolean>,>
      <hidden : <boolean>,>
      <priority: <priority>,>
      <tags: { <document> },>
      <slaveDelay : <number>,>
      <votes : <number>>
    }
    , ...
  ],
  <settings: {
    <getLastErrorDefaults : <lasterrdefaults>,>
    <chainingAllowed : <boolean>,>
    <getLastErrorModes : <modes>>
  }>
}
```

Configuration Variables

`local.system.replset._id`

Type: string

Value: <setname>

An `_id` field holding the name of the replica set. This reflects the set name configured with `replSet` (page 1124) or *mongod --replSet*.

`local.system.replset.members`

Type: array

Contains an array holding an embedded *document* for each member of the replica set. The `members` document contains a number of fields that describe the configuration of each member of the replica set.

The `members` (page 474) field in the replica set configuration document is a zero-indexed array.

`local.system.replset.members[n]._id`

Type: ordinal

Provides the zero-indexed identifier of every member in the replica set.

Note: When updating the replica configuration object, access the replica set members in the `members` (page 474) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 474) field in each document in the `members` (page 474) array.

`local.system.replset.members[n].host`

Type: <hostname><:port>

Identifies the host name of the set member with a hostname and port number. This name must be resolvable for every host in the replica set.

Warning: `host` (page 474) cannot hold a value that resolves to `localhost` or the local interface unless *all* members of the set are on hosts that resolve to localhost.

`local.system.replset.members[n].arbiterOnly`

Optional.

Type: boolean

Default: false

Identifies an arbiter. For arbiters, this value is `true`, and is automatically configured by `rs.addArb()` (page 1015)”.

`local.system.replset.members[n].buildIndexes`

Optional.

Type: boolean

Default: true

Determines whether the `mongod` (page 1049) builds *indexes* on this member. Do not set to `false` if a replica set *can* become a primary, 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` (page 1075),
- this instance will receive no queries, *and*
- index creation and maintenance overburdens the host system.

If set to `false`, secondaries configured with this option *do* build indexes on the `_id` field, to facilitate operations required for replication.

> **Warning:** You may only set this value when adding a member to a replica set. You may not reconfigure a replica set to change the value of the `buildIndexes` (page 475) field after adding the member to the set. Other secondaries cannot replicate from a members where `buildIndexes` (page 475) is false.

`local.system.replset.members[n].hidden`

Optional.

Type: boolean

Default: false

When this value is `true`, the replica set hides this instance, and does not include the member in the output of `db.isMaster()` (page 1008) or `isMaster` (page 871). 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 379)“

`local.system.replset.members[n].priority`

Optional.

Type: Number, between 0 and 100.0 including decimals.

Default: 1

Specify higher values to make a member *more* eligible to become *primary*, and lower values to make the member *less* eligible to become primary. Priorities are only used in comparison to each other. Members of the set will veto election requests from members when another eligible member has a higher priority value. Changing the balance of priority in a replica set will trigger an election.

A `priority` (page 475) of `0` makes it impossible for a member to become primary.

See also:

“`priority` (page 475)” and “*Replica Set Elections* (page 389).”

`local.system.replset.members[n].`**`tags`**

Optional.

Type: *MongoDB Document*

Default: none

Used to represent arbitrary values for describing or tagging members for the purposes of extending *write concern* to allow configurable data center awareness.

Use in conjunction with `getLastErrorModes` (page 477) and `getLastErrorDefaults` (page 477) and `db.getLastError()` (page 1005) (i.e. `getLastError` (page 861).)

For procedures on configuring tag sets, see *Configure Replica Set Tag Sets* (page 444).

Important: In tag sets, all tag values must be strings.

`local.system.replset.members[n].`**`slaveDelay`**

Optional.

Type: Integer. (seconds.)

Default: 0

Describes the number of seconds "behind" the primary that this replica set member should "lag." Use this option to create *delayed members* (page 379), that maintain a copy of the data that reflects the state of the data set at some amount of time in the past, specified in seconds. Typically such delayed members help protect against human error, and provide some measure of insurance against the unforeseen consequences of changes and updates.

`local.system.replset.members[n].`**`votes`**

Optional.

Type: Integer

Default: 1

Controls the number of votes a server will cast in a *replica set election* (page 389). The number of votes each member has can be any non-negative integer, but it is highly recommended each member has 1 or 0 votes.

If you need more than 7 members in one replica set, use this setting to add additional non-voting members with a `votes` (page 476) value of `0`.

For most deployments and most members, use the default value, `1`, for `votes` (page 476).

`local.system.replset.`**`settings`**

Optional.

Type: *MongoDB Document*

The `settings` document configures options that apply to the whole replica set.

`local.system.replset.settings.`**`chainingAllowed`**

Optional.

Type: boolean

Default: true

New in version 2.2.4.

When `chainingAllowed` (page 476) is `true`, the replica set allows *secondary* members to replicate from other secondary members. When `chainingAllowed` (page 476) is `false`, secondaries can replicate only from the *primary*.

When you run `rs.config()` (page 1015) to view a replica set's configuration, the `chainingAllowed` (page 476) field appears only when set to `false`. If not set, `chainingAllowed` (page 476) is `true`.

See also:

Manage Chained Replication (page 450)

`local.system.replset.settings.`**`getLastErrorDefaults`**

Optional.

Type: *MongoDB Document*

Specify arguments to the `getLastError` (page 861) that members of this replica set will use when no arguments to `getLastError` (page 861) has no arguments. If you specify *any* arguments, `getLastError` (page 861) , ignores these defaults.

`local.system.replset.settings.`**`getLastErrorModes`**

Optional.

Type: *MongoDB Document*

Defines the names and combination of `members` (page 474) for use by the application layer to guarantee *write concern* to database using the `getLastError` (page 861) command to provide *data-center awareness*.

Example Reconfiguration Operations

Most modifications of *replica set* configuration use the `mongo` (page 1066) shell. Consider the following reconfiguration operation:

Example

Given the following replica set configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
                   {
                      "_id" : 0,
                      "host" : "mongodb0.example.net:27017"
                   },
                   {
                      "_id" : 1,
                      "host" : "mongodb1.example.net:27017"
                   },
                   {
                      "_id" : 2,
                      "host" : "mongodb2.example.net:27017"
                   }
     ]
}
```

The following reconfiguration operation updates the `priority` (page 475) of the replica set members:

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

First, this operation sets the local variable `cfg` to the current replica set configuration using the `rs.conf()` (page 1015) method. Then it adds priority values to the `cfg` *document* for the three sub-documents in the `members` (page 474) array, accessing each replica set member with the array index and **not** the replica set member's `_id` (page 474) field. Finally, it calls the `rs.reconfig()` (page 1016) method with the argument of `cfg` to initialize this new configuration. The replica set configuration after this operation will resemble the following:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
                   {
                      "_id" : 0,
                      "host" : "mongodb0.example.net:27017",
                      "priority" : 0.5
                   },
                   {
                      "_id" : 1,
                      "host" : "mongodb1.example.net:27017",
                      "priority" : 2
                   },
                   {
                      "_id" : 2,
                      "host" : "mongodb2.example.net:27017",
                      "priority" : 1
                   }
     ]
}
```

Using the "dot notation" demonstrated in the above example, you can modify any existing setting or specify any of optional *replica set configuration variables* (page 474). Until you run `rs.reconfig(cfg)` at the shell, no changes will take effect. You can issue `cfg = rs.conf()` at any time before using `rs.reconfig()` (page 1016) to undo your changes and start from the current configuration. If you issue `cfg` as an operation at any point, the `mongo` (page 1066) shell at any point will output the complete *document* with modifications for your review.

The `rs.reconfig()` (page 1016) 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()` (page 1016) can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

Note: The `rs.reconfig()` (page 1016) shell method can force the current primary to step down and triggers an election in some situations. When the primary steps down, all clients will disconnect. This is by design. Since this typically takes 10-20 seconds, attempt to make such changes during scheduled maintenance periods.

30.3.3 The `local` Database

Overview

Every `mongod` (page 1049) instance has its own `local` database, which stores data used in the replication process, and other instance-specific data. The `local` database is invisible to replication: collections in the `local` database are not replicated.

In replication, the `local` database store stores internal replication data for each member of a *replica set*. The `local` stores the following collections:

Changed in version 2.4: When running with authentication (i.e. `auth` (page 1118)), authenticating to the `local` database is **not** equivalent to authenticating to the `admin` database. In previous versions, authenticating to the `local` database provided access to all databases.

Collection on all `mongod` Instances

`local.`**`startup_log`**

On startup, each `mongod` (page 1049) instance inserts a document into `startup_log` (page 479) with diagnostic information about the `mongod` (page 1049) instance itself and host information. `startup_log` (page 479) is a capped collection. This information is primarily useful for diagnostic purposes.

Example

Consider the following prototype of a document from the `startup_log` (page 479) collection:

```
{
  "_id" : "<string>",
  "hostname" : "<string>",
  "startTime" : ISODate("<date>"),
  "startTimeLocal" : "<string>",
  "cmdLine" : {
        "dbpath" : "<path>",
        "<option>" : <value>
  },
  "pid" : <number>,
  "buildinfo" : {
        "version" : "<string>",
        "gitVersion" : "<string>",
        "sysInfo" : "<string>",
        "loaderFlags" : "<string>",
        "compilerFlags" : "<string>",
        "allocator" : "<string>",
        "versionArray" : [ <num>, <num>, <...> ],
        "javascriptEngine" : "<string>",
        "bits" : <number>,
        "debug" : <boolean>,
        "maxBsonObjectSize" : <number>
  }
}
```

Documents in the `startup_log` (page 479) collection contain the following fields:

`local.startup_log.`**`_id`**

Includes the system hostname and a millisecond epoch value.

`local.startup_log.`**`hostname`**

The system's hostname.

`local.startup_log.`**`startTime`**

A UTC *ISODate* value that reflects when the server started.

`local.startup_log.`**`startTimeLocal`**

A string that reports the `startTime` (page 479) in the system's local time zone.

`local.startup_log.`**`cmdLine`**

A sub-document that reports the `mongod` (page 1049) runtime options and their values.

`local.startup_log.pid`
The process identifier for this process.

`local.startup_log.buildinfo`
A sub-document that reports information about the build environment and settings used to compile this `mongod` (page 1049). This is the same output as `buildInfo` (page 899). See `buildInfo` (page 899).

Collections on Replica Set Members

`local.system.replset`
`local.system.replset` (page 480) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` (page 1015) from the `mongo` (page 1066) shell. You can also query this collection directly.

`local.oplog.rs`
`local.oplog.rs` (page 480) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSize` (page 1124) setting. To resize the oplog after replica set initiation, use the *Change the Size of the Oplog* (page 439) procedure. For additional information, see the *Oplog Size* (page 405) section.

`local.replset.minvalid`
This contains an object used internally by replica sets to track replication status.

`local.slaves`
This contains information about each member of the set and the latest point in time that this member has synced to. If this collection becomes out of date, you can refresh it by dropping the collection and allowing MongoDB to automatically refresh it during normal replication:

```
db.getSiblingDB("local").slaves.drop()
```

Collections used in Master/Slave Replication

In *master*/*slave* replication, the `local` database contains the following collections:

- On the master:

  `local.oplog.$main`
  This is the oplog for the master-slave configuration.

  `local.slaves`
  This contains information about each slave.

- On each slave:

  `local.sources`
  This contains information about the slave's master server.

30.3.4 Replica Set Member States

Members of replica sets have states that reflect the startup process, basic operations, and potential error states.

Number	Name	State Description
0	STARTUP (page 481)	Cannot vote. All members start up in this state. The mongod (page 1049) parses the *replica set configuration document* (page 431) while in STARTUP (page 481).
1	PRIMARY (page 481)	Can vote. The *primary* (page 374) is the only member to accept write operations.
2	SECONDARY (page 481)	Can vote. The *secondary* (page 374) replicates the data store.
3	RECOVERING (page 482)	Can vote. Members either perform startup self-checks, or transition from completing a *rollback* (page 393) or *resync* (page 443).
4	FATAL (page 482)	Cannot vote. Has encountered an unrecoverable error.
5	STARTUP2 (page 481)	Cannot vote. Forks replication and election threads before becoming a secondary.
6	UNKNOWN (page 482)	Cannot vote. Has never connected to the replica set.
7	ARBITER (page 481)	Can vote. *Arbiters* (page **??**) do not replicate data and exist solely to participate in elections.
8	DOWN (page 482)	Cannot vote. Is not accessible to the set.
9	ROLLBACK (page 482)	Can vote. Performs a *rollback* (page 393).
10	SHUNNED (page 482)	Cannot vote. Was once in the replica set but has now been removed.

States

Core States

PRIMARY
Members in PRIMARY (page 481) state accept write operations. A replica set has only one primary at a time. A SECONDARY (page 481) member becomes primary after an *election* (page 389). Members in the PRIMARY (page 481) state are eligible to vote.

SECONDARY
Members in SECONDARY (page 481) state replicate the primary's data set and can be configured to accept read operations. Secondaries are eligible to vote in elections, and may be elected to the PRIMARY (page 481) state if the primary becomes unavailable.

ARBITER
Members in ARBITER (page 481) state do not replicate data or accept write operations. They are eligible to vote, and exist solely to break a tie during elections. Replica sets should only have a member in the ARBITER (page 481) state if the set would otherwise have an even number of members, and could suffer from tied elections. Like primaries, there should only be at most one arbiter in any replica set.

See *Replica Set Members* (page 374) for more information on core states.

Initialization States

STARTUP
Each member of a replica set starts up in STARTUP (page 481) state. mongod (page 1049) then loads that member's replica set configuration, and transitions the member's state to STARTUP2 (page 481). Members in STARTUP (page 481) are not eligible to vote.

STARTUP2

Each member of a replica set enters the `STARTUP2` (page 481) state as soon as `mongod` (page 1049) finishes loading that member's configuration. While in the `STARTUP2` (page 481) state, the member creates threads to handle internal replication operations. Members are in the `STARTUP2` (page 481) state for a short period of time before entering the `RECOVERING` (page 482) state. Members in the `STARTUP2` (page 481) state are not eligible to vote.

RECOVERING

A member of a replica set enters `RECOVERING` (page 482) state when it is not ready to accept reads. The `RECOVERING` (page 482) state can occur during normal operation, and doesn't necessarily reflect an error condition. Members in the `RECOVERING` (page 482) state are eligible to vote in elections, but is not eligible to enter the `PRIMARY` (page 481) state.

During startup, members transition through `RECOVERING` (page 482) after `STARTUP2` (page 481) and before becoming `SECONDARY` (page 481).

During normal operation, if a *secondary* falls behind the other members of the replica set, it may need to *resync* (page 443) with the rest of the set. While resyncing, the member enters the `RECOVERING` (page 482) state.

Whenever the replica set replaces a *primary* in an election, the old primary's data collection may contain documents that did not have time to replicate to the *secondary* members. In this case the member rolls back those writes. During *rollback* (page 393), the member will have `RECOVERING` (page 482) state.

On secondaries, the `compact` (page 890) and `replSetMaintenance` (page 867) commands force the secondary to enter `RECOVERING` (page 482) state. This prevents clients from reading during those operations.

Error States

Members in any error state can't vote.

FATAL

Members that encounter an unrecoverable error enter the `FATAL` (page 482) state. Members in this state requires administrator intervention.

UNKNOWN

Members that have never communicated status information to the replica set are in the `UNKNOWN` (page 482) state.

DOWN

Members that lose their connection to the replica set enter the `DOWN` (page 482) state.

SHUNNED

Members that are removed from the replica set enter the `SHUNNED` (page 482) state.

ROLLBACK

When a `SECONDARY` (page 481) rolls back a write operation after transitioning from `PRIMARY` (page 481), it enters the `ROLLBACK` (page 482) state. See *Rollbacks During Replica Set Failover* (page 393).

Part IX

Sharding

Sharding is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

CHAPTER 31

Sharding Introduction

Sharding is a method for storing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

31.1 Purpose of Sharding

Database systems with large data sets and high throughput applications can challenge the capacity of a single server. High query rates can exhaust the CPU capacity of the server. Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

To address these issues of scales, database systems have two basic approaches: **vertical scaling** and **sharding**.

Vertical scaling adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately *more expensive* than smaller systems. Additionally, cloud-based providers may only allow users to provision smaller instances. As a result there is a *practical maximum* capability for vertical scaling.

Sharding, or *horizontal scaling*, by contrast, divides the data set and distributes the data over multiple servers, or **shards**. Each shard is an independent database, and collectively, the shards make up a single logical database.

Sharding addresses the challenge of scaling to support high throughout and large data sets:

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, shared clusters can increase capacity and throughput *horizontally*.

  For example, to insert data, the application only needs to access the shard responsible for that records.

- Sharding reduces the amount of data that each server needs to store. Each shard stores less data as the cluster grows.

  For example, if a database has a 1 terabyte data set, and there are 4 shards, then each shard might hold only 256GB of data. If there are 40 shards, then each shard might hold only 25GB of data.

31.2 Sharding in MongoDB

MongoDB supports sharding through the configuration of a *sharded clusters*.

Sharded cluster has the following components: *shards*, *query routers* and *config servers*.

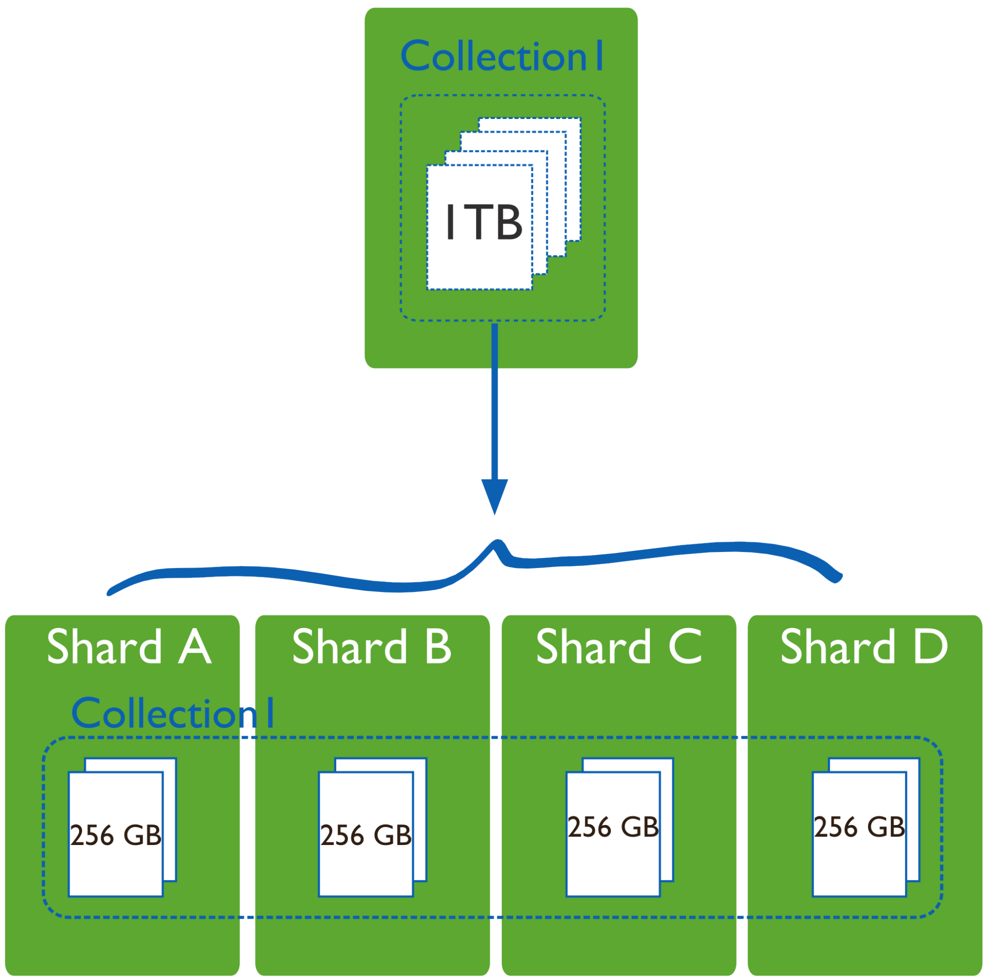


Figure 31.1: Diagram of a large collection with data distributed across 4 shards.

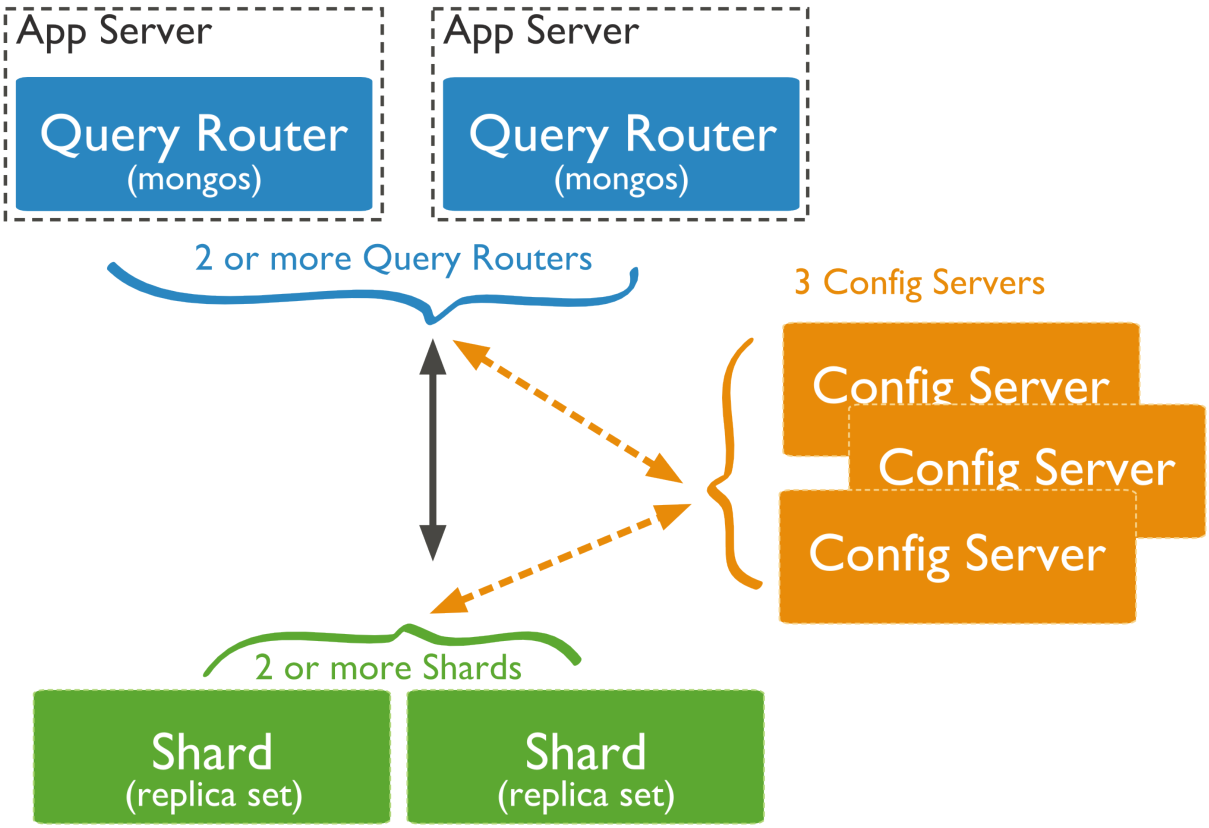


Figure 31.2: Diagram of a sample sharded cluster for production purposes. Contains exactly 3 config server, 2 or more `mongos` (page 1061) query router, and at least 2 shards. The shards are replica sets.

Shards store the data. To provide high availability and data consistency, in a production sharded cluster, each shard is a *replica set* [1]. For more information on replica sets, see *Replica Sets* (page 373).

Query Routers, or `mongos` (page 1061) instances, interface with client applications and direct operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Most sharded cluster have many query routers.

Config servers store the cluster's metadata. This data contains a mapping of the cluster's data set to the shards. The query router uses this metadata to target operations to specific shards. Production sharded clusters have *exactly* 3 config servers.

31.3 Data Partitioning

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data by the **shard key**.

31.3.1 Shard Keys

To shard a collection, you need to select a **shard key**. A *shard key* is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB divides the shard key values into **chunks** and distributes the *chunks* evenly across the shards. To divide the shard key values into chunks, MongoDB uses either **range based partitioning** and **hash based partitioning**. See *Shard Keys* (page 502) for more information.

31.3.2 Range Based Sharding

For *range-based sharding*, MongoDB divides the data set into ranges determined by the shard key values to provide **range based partitioning**. Consider a numeric shard key: If you visualize a number line that goes from negative infinity to positive infinity, each value of the shard key falls at some point on that line. MongoDB partitions this line into smaller, non-overlapping ranges called **chunks** where a chunk is range of values from some minimum value to some maximum value.

Given a range based partitioning system, documents with "close" shard key values are likely to be in the same chunk, and therefore on the same shard.

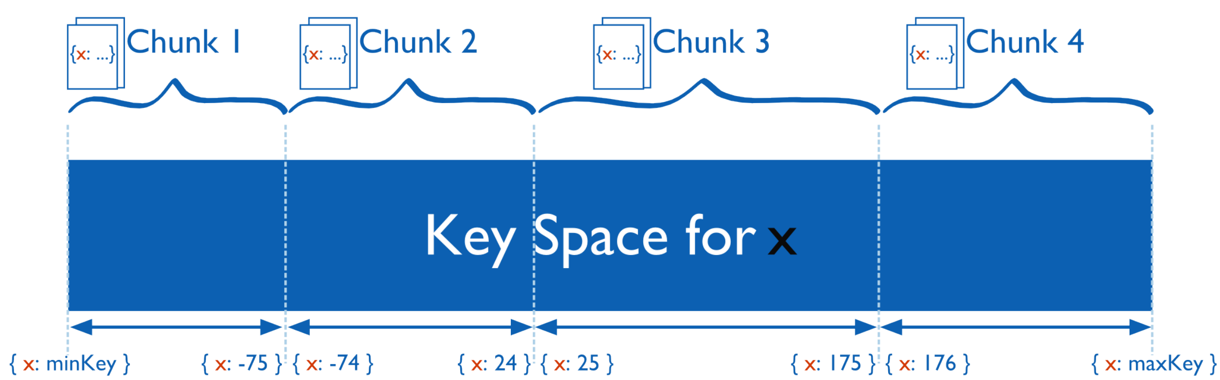


Figure 31.3: Diagram of the shard key value space segmented into smaller ranges or chunks.

[1] For development and testing purposes only, each **shard** can be a single `mongod` (page 1049) instead of a replica set. Do **not** deploy production clusters without 3 config servers.

31.3.3 Hash Based Sharding

For *hash based partitioning*, MongoDB computes a hash of a field's value, and then uses these hashes to create chunks.

With hash based partitioning, two documents with "close" shard key values are *unlikely* to be part of the same chunk. This ensures a more random distribution of a collection in the cluster.

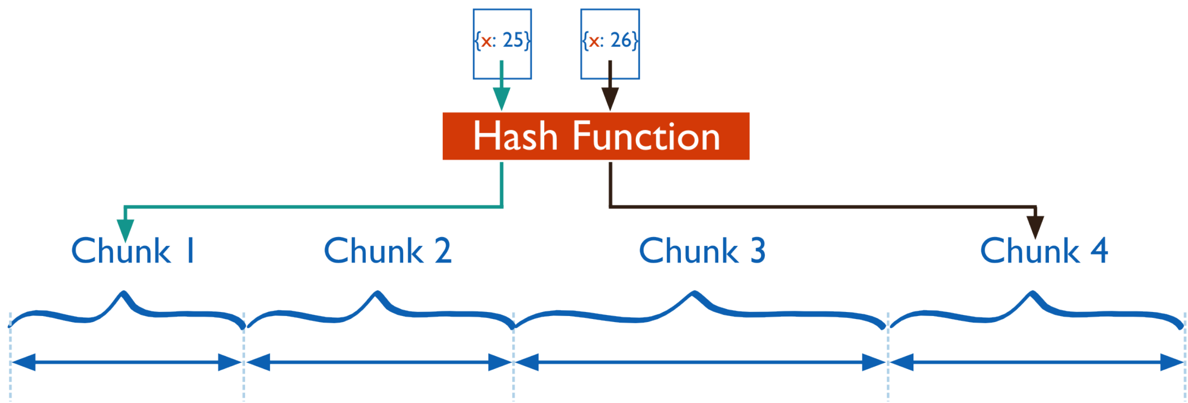


Figure 31.4: Diagram of the hashed based segmentation.

31.3.4 Performance Distinctions between Range and Hash Based Partitioning

Range based partitioning supports more efficient range queries. Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks.

However, range based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding. For example, if the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard. In this situation, a small set of shards may receive the majority of requests and the system would not scale very well.

Hash based partitioning, by contrast, ensures an even distribution of data at the expense of efficient range queries. Hashed key values results in random distribution of data across chunks and therefore shards. But random distribution makes it more likely that a range query on the shard key will not be able to target a few shards but would more likely query every shard in order to return a result.

31.4 Maintaining a Balanced Data Distribution

The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster, such as a particular shard contains significantly more chunks than another shard or a size of a chunk is significantly greater than other chunk sizes.

MongoDB ensures a balanced cluster using two background process: splitting and the balancer.

31.4.1 Splitting

Splitting is a background process that keeps chunks from growing too large. When a chunk grows beyond a *specified chunk size* (page 512), MongoDB splits the chunk in half. Inserts and updates triggers splits. Splits are a efficient meta-data change. To create splits, MongoDB does *not* migrate any data or affect the shards.

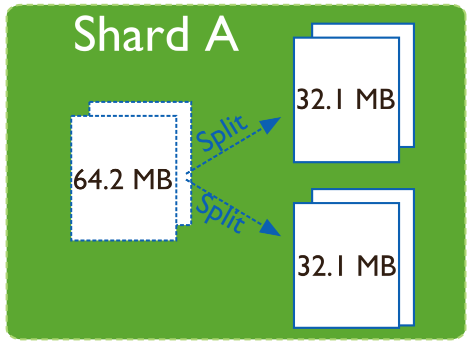


Figure 31.5: Diagram of a shard with a chunk that exceeds the default *chunk size* (page 512) of 64 MB and triggers a split of the chunk into two chunks.

31.4.2 Balancing

The *balancer* (page 510) is a background process that manages chunk migrations. The balancer runs in all of the query routers in a cluster.

When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances. For example: if collection `users` has 100 chunks on *shard 1* and 50 chunks and *shard 2*, the balancer will migrate chunks from *shard 1* to *shard 2* until the collections achieves balance.

The shards manage *chunk migrations* as a background operation. During migration, all requests for a chunks data address the "origin" shard.

In a chunk migration, the *destination shard* receives all the documents in the chunk from the *origin shard*. Then, the destination shard captures and applies all changes made to the data during migration process. Finally, the destination shard updates the metadata regarding the location of the on *config server*.

If there's an error during the migration, the balancer aborts the process leaving the chunk on the origin shard. MongoDB removes the chunks data from the origin shard **after** the migration completes successfully.

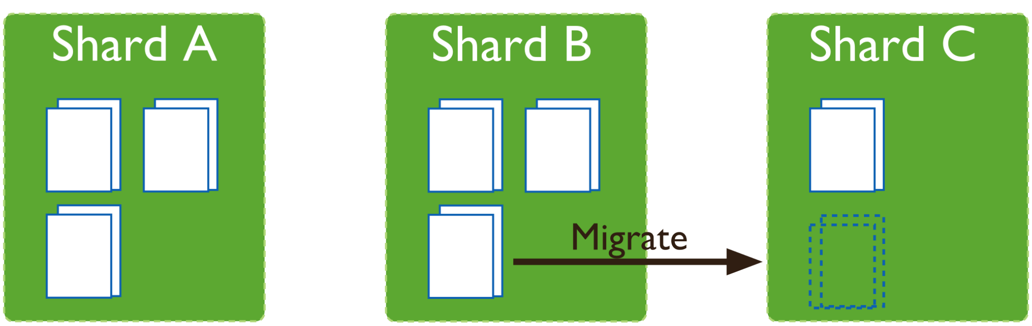


Figure 31.6: Diagram of a collection distributed across three shards. For this collection, the difference in the number of chunks between the shards reaches the *migration thresholds* (page 510) (in this case, 2) and triggers migration.

31.4.3 Adding and Removing Shards from the Cluster

Adding a shard to a cluster creates an imbalance since the new shard has no chunks. While MongoDB begins migrating data to the new shard immediately, it can take some time before the cluster balances.

When removing a shard, the balancer migrates all chunks from to other shards. After migrating all data and updating the meta data, you can safely remove the shard.

CHAPTER 32

Sharding Concepts

These documents present the details of sharding in MongoDB. These include the components, the architectures, and the behaviors of MongoDB sharded clusters. For an overview of sharding and sharded clusters, see *Sharding Introduction* (page 487).

32.1 Sharded Cluster Components

Sharded clusters implement *sharding*. A sharded cluster consists of the following components:

Shards A shard is a MongoDB instance that holds a subset of a collection's data. Each shard is either a single `mongod` (page 1049) instance or a *replica set*. In production, all shards are replica sets. For more information see *Shards* (page 496).

Config Servers Each *config server* (page 498) is a `mongod` (page 1049) instance that holds metadata about the cluster. The metadata maps *chunks* to shards. For more information, see *Config Servers* (page 498).

Routing Instances Each router is a `mongos` (page 1061) instance that routes the reads and writes from applications to the shards. Applications do not access the shards directly. For more information see *Sharded Cluster Query Routing* (page 506).

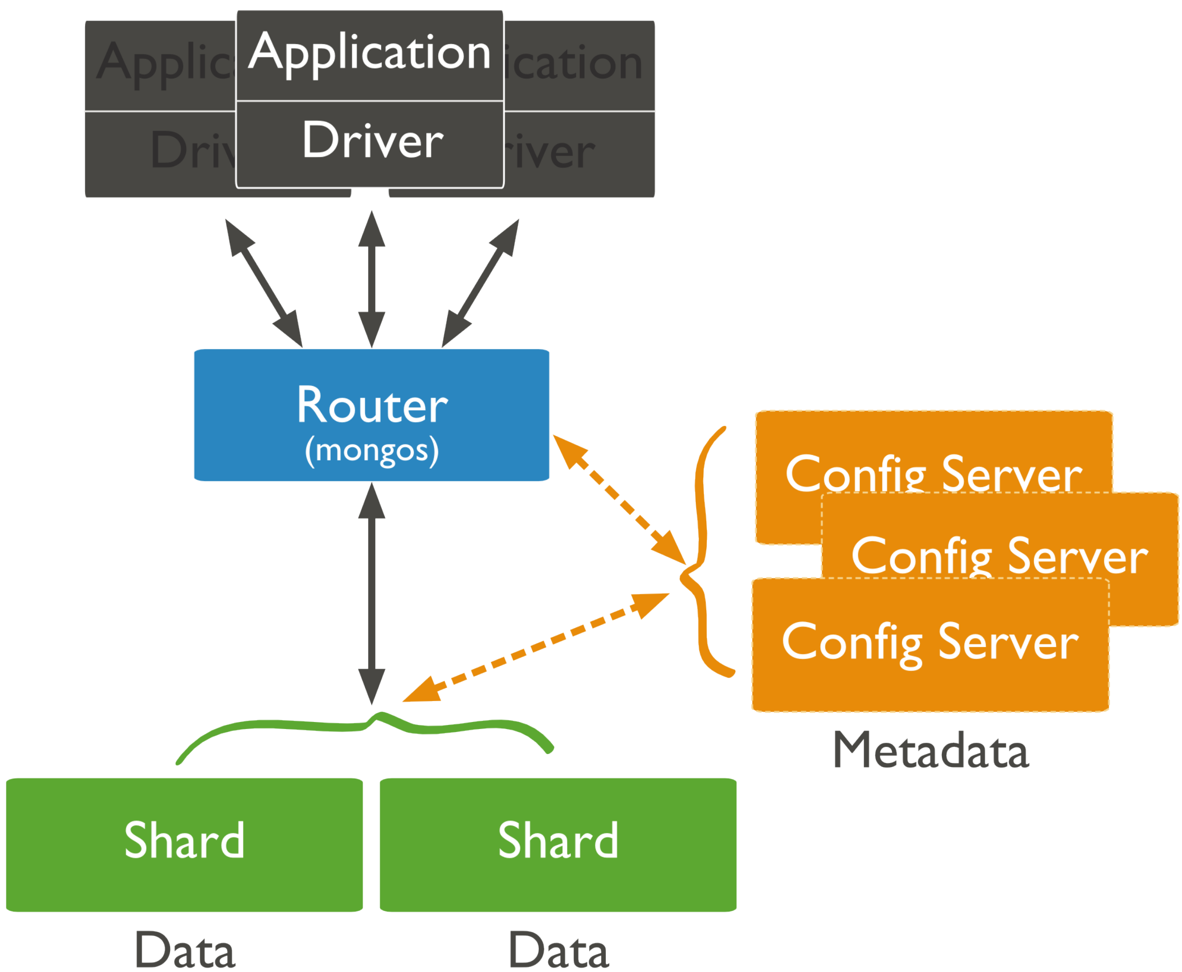


Figure 32.1: Diagram of a sharded cluster.

Enable sharding in MongoDB on a per-collection basis. For each collection you shard, you will specify a *shard key* for that collection.

Deploy a sharded cluster, see *Deploy a Sharded Cluster* (page 516).

32.1.1 Shards

A shard is a *replica set* or a single `mongod` (page 1049) that contains a subset of the data for the sharded cluster. Together, the cluster's shards hold the entire data set for the cluster.

Typically each shard is a replica set. The replica set provides redundancy and high availability for the data in each shard.

Important: MongoDB shards data on a *per collection* basis. You *must* access all data in a sharded cluster via the `mongos` (page 1061) instances. If you connect directly to a shard, you will see only its fraction of the cluster's data. There is no particular order to the data set on a specific shard. MongoDB does not guarantee that any two contiguous chunks will reside on a single shard.

Primary Shard

Every database has a "primary" [1] shard that holds all the un-sharded collections in that database.

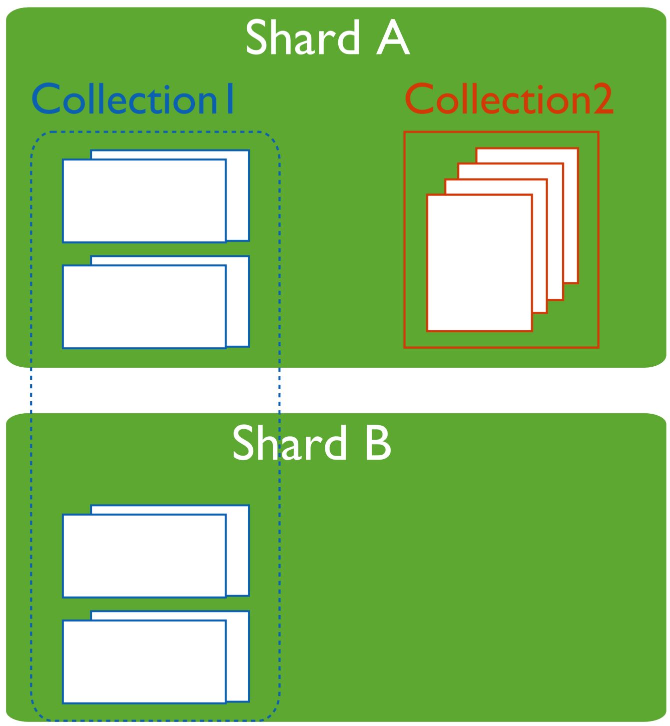


Figure 32.2: Diagram of a primary shard. A primary shard contains non-sharded collections as well as chunks of documents from sharded collections. Shard A is the primary shard.

To change the primary shard for a database, use the `movePrimary` (page 881) command.

Warning: The `movePrimary` (page 881) command can be expensive because it copies all non-sharded data to the new shard. During this time, this data will be unavailable for other operations.

When you deploy a new *sharded cluster*, the "first" shard becomes the primary shard for all existing databases before enabling sharding. Databases created subsequently may reside on any shard in the cluster.

[1] The term "primary" shard has nothing to do with the term *primary* in the context of *replica sets*.

Shard Status

Use the `sh.status()` (page 1029) method in the `mongo` (page 1066) shell to see an overview of the cluster. This reports includes which shard is primary for the database and the *chunk* distribution across the shards. See `sh.status()` (page 1029) method for more details.

32.1.2 Config Servers

Config servers are special `mongod` (page 1049) instances that store the *metadata* (page 513) for a sharded cluster. Config servers use a two-phase commit to ensure immediate consistency and reliability. Config servers *do not* run as replica sets. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata.

A production sharded cluster has *exactly three* config servers. For testing purposes you may deploy a cluster with a single config server. But to ensure redundancy and safety in production, you should always use three.

> **Warning:** If your cluster has a single config server, then the config server is a single point of failure. If the config server is inaccessible, the cluster is not accessible. If you cannot recover the data on a config server, the cluster will be inoperable.
> **Always** use three config servers for production deployments.

Config servers store metadata for a single sharded cluster. Each cluster must have its own config servers.

Config Database

Config servers store the metadata in the *config database* (page 555). The `mongos` (page 1061) instances cache this data and use it to route reads and writes to shards.

Read and Write Operations on Config Servers

The load on the config servers is small because each `mongos` (page 1061) instance caches the metadata.

MongoDB only writes data to the config server in the following cases:

- To create splits in existing chunks. For more information, see *chunk splitting* (page 512).
- To migrate a chunk between shards. For more information, see *chunk migration* (page 511).

MongoDB only reads data from the config server data in the following cases:

- A new `mongos` (page 1061) starts for the first time, or an existing `mongos` (page 1061) restarts.
- After a chunk migration, the `mongos` (page 1061) instances update themselves with the new cluster metadata.

Config Server Availability

If one or two config servers become unavailable, the cluster's metadata becomes *read only*. You can still read and write data from the shards, but no chunk migrations or splits will occur until all three servers are available.

If all three config servers are unavailable, you can still use the cluster if you do not restart the `mongos` (page 1061) instances until after the config servers are accessible again. If you restart the `mongos` (page 1061) instances before the config servers are available, the `mongos` (page 1061) will be unable to route reads and writes.

Clusters become inoperable without the cluster metadata. *Always,* ensure that the config servers remain available and intact. As such, backups of config servers are critical. The data on the config server is small compared to the data stored in a cluster. This means the config server has a relatively low activity load, and the config server does not need to be always available to support a sharded cluster. As a result, it is easy to back up the config servers.

32.2 Sharded Cluster Architectures

The following documents introduce deployment patterns for sharded clusters.

Sharded Cluster Requirements **(page 499)** Discusses the requirements for sharded clusters in MongoDB.

Production Cluster Architecture **(page 500)** Sharded cluster for production has component requirements to provide redundancy and high availability.

Sharded Cluster Test Architecture **(page 500)** Sharded clusters for testing and development can have fewer components.

32.2.1 Sharded Cluster Requirements

While sharding is a powerful and compelling, sharded clusters have significant infrastructure requirements and increases the overall complexity of a deployment. As a result, only deploy sharded clusters when indicated by application and operational requirements

Sharding is the *only* solution for some classes of deployments. Use *sharded clusters* if:

- your data set approaches or exceeds the storage capacity of a single MongoDB instance.
- the size of your system's active *working set* *will soon* exceed the capacity of your system's *maximum* RAM.
- a single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add complexity to your system without adding much benefit.

Important: It takes time and resources to deploy sharding. If your system has *already* reached or exceeded its capacity, it will be difficult to deploy sharding without impacting your application.

As a result, if you think you will need to partition your database in the future, **do not** wait until your system is overcapacity to enable sharding.

When designing your data model, take into consideration your sharding needs.

Data Quantity Requirements

Your cluster should manage a large quantity of data if sharding is to have an effect. The default *chunk* size is 64 megabytes. And the *balancer* (page 510) will not begin moving data across shards until the imbalance of chunks among the shards exceeds the *migration threshold* (page 510). In practical terms, unless your cluster has many hundreds of megabytes of data, your data will remain on a single shard.

In some situations, you may need to shard a small collection of data. But most of the time, sharding a small collection is not worth the added complexity and overhead unless you need additional write capacity. If you have a small data set, a properly configured single MongoDB instance or a replica set will usually be enough for your persistence layer needs.

Chunk size is `user configurable`. For most deployments, the default value is of 64 megabytes is ideal. See *Chunk Size* (page 512) for more information.

32.2.2 Production Cluster Architecture

In a production cluster, you must ensure that data is redundant and that your systems are highly available. To that end, a production cluster must have the following components:

- Three *config servers* (page 498). Each config servers must be on separate machines. A single *sharded cluster* must have exclusive use of its *config servers* (page 498). If you have multiple sharded clusters, you will need to have a group of config servers for each cluster.
- Two or more *replica sets*. These replica sets are the *shards*. For information on replica sets, see *Replication* (page 367).
- One or more `mongos` (page 1061) instances. `mongos` (page 1061) is the routers for the cluster. Typically, deployments have one `mongos` (page 1061) instance on each application server. You may also may deploy a group of `mongos` (page 1061) instances and use a proxy/load balancer between the application and the `mongos` (page 1061).

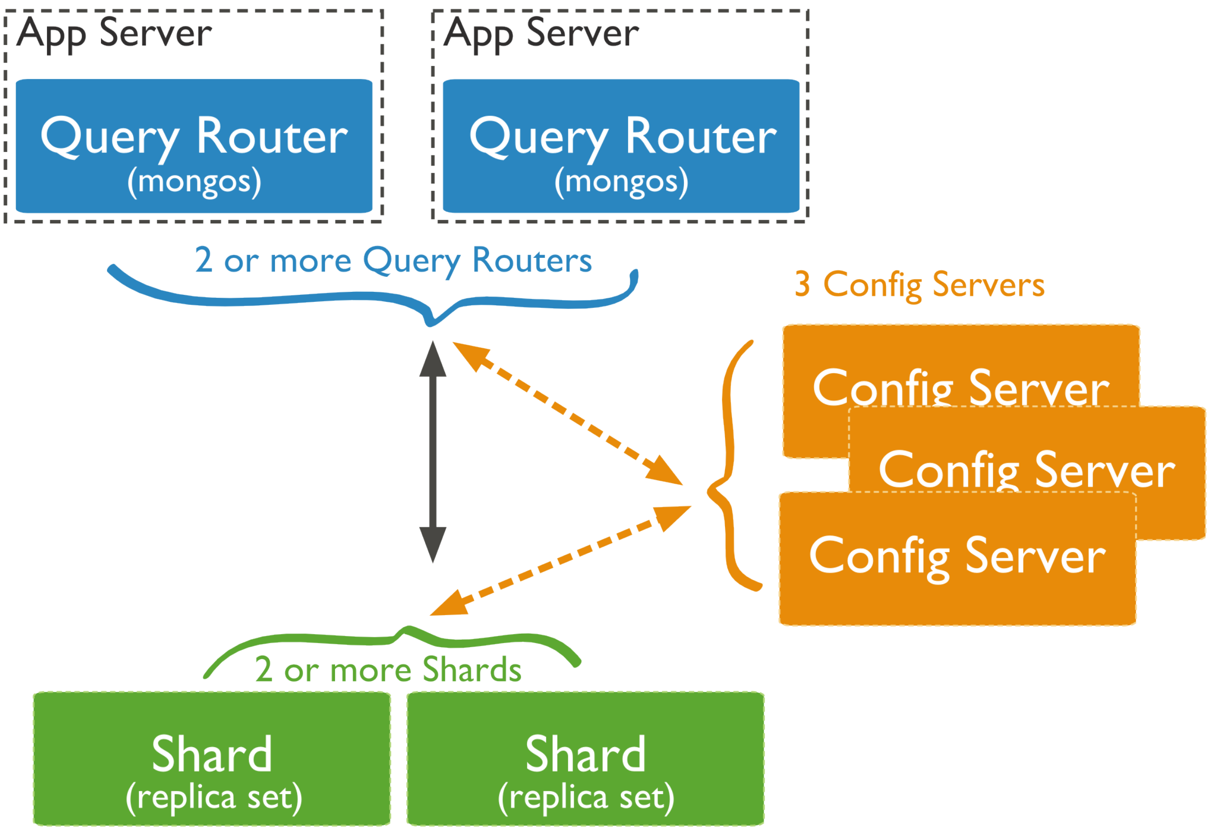


Figure 32.3: Diagram of a sample sharded cluster for production purposes. Contains exactly 3 config server, 2 or more `mongos` (page 1061) query router, and at least 2 shards. The shards are replica sets.

32.2.3 Sharded Cluster Test Architecture

Warning: Use the test cluster architecture for testing and development only.

For testing and development, you can deploy a minimal sharded clusters cluster. These **non-production** clusters have the following components:

- One *config server* (page 498).
- At least one shard. Shards are either *replica sets* or a standalone `mongod` (page 1049) instances.
- One `mongos` (page 1061) instance.

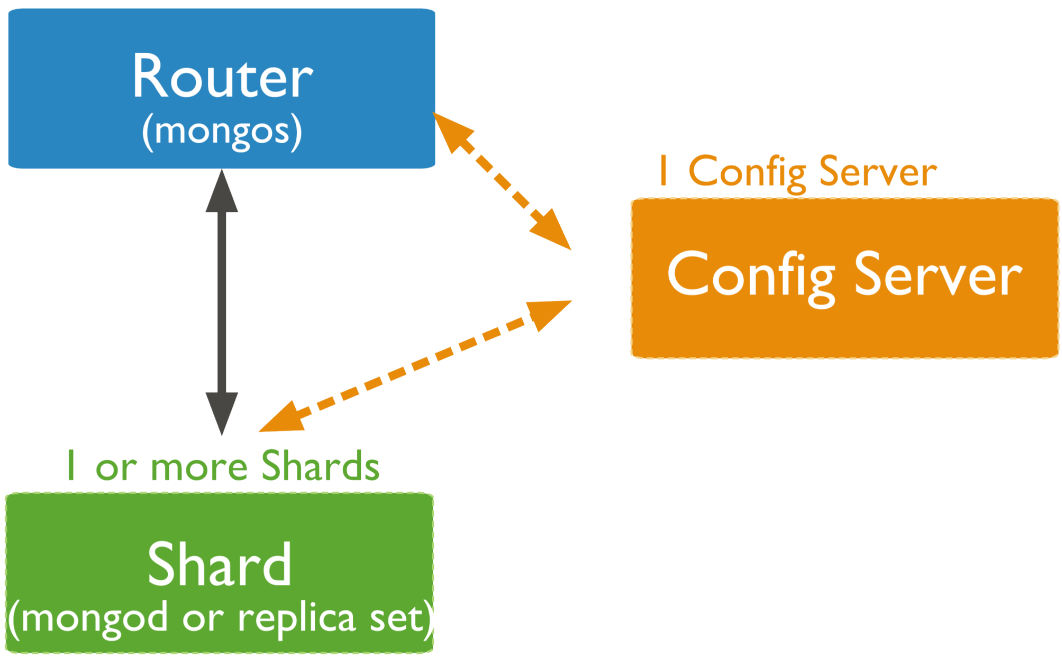


Figure 32.4: Diagram of a sample sharded cluster for testing/development purposes only. Contains only 1 config server, 1 `mongos` (page 1061) router, and at least 1 shard. The shard can be either a replica set or a standalone `mongod` (page 1049) instance.

See

Production Cluster Architecture (page 500)

32.3 Sharded Cluster Behavior

These documents address the distribution of data and queries to a sharded cluster as well as specific security and availability considerations for sharded clusters.

Shard Keys **(page 502)** MongoDB uses the shard key to divide a collection's data across the cluster's shards.

Sharded Cluster High Availability **(page 504)** Sharded clusters provide ways to address some availability concerns.

Sharded Cluster Security **(page 505)** MongoDB controls access to sharded clusters with key files.

Sharded Cluster Query Routing **(page 506)** The cluster's routers, or `mongos` instances, send reads and writes to the relevant shard or shards.

32.3.1 Shard Keys

The shard key determines the distribution of the collection's *documents* among the cluster's *shards*. The shard key is either an indexed *field* or an indexed compound field that exists in every document in the collection.

MongoDB partitions data in the collection using ranges of shard key values. Each range, or *chunk*, defines a non-overlapping range of shard key values. MongoDB distributes the chunks, and their documents, among the shards in the cluster.

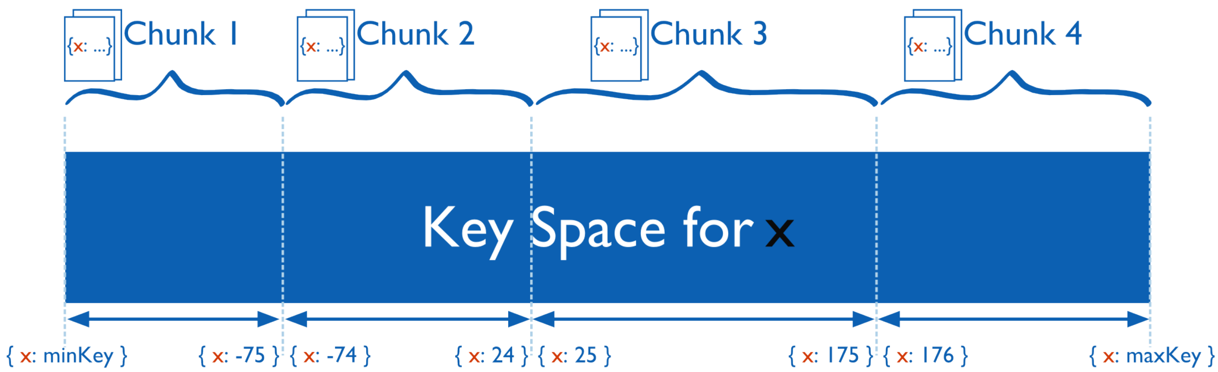


Figure 32.5: Diagram of the shard key value space segmented into smaller ranges or chunks.

When a chunk grows beyond the *chunk size* (page 512), MongoDB *splits* the chunk into smaller chunks, always based on ranges in the shard key.

Hashed Shard Keys

New in version 2.4.

Hashed shard keys use a *hashed index* (page 332) of a single field as the *shard key* to partition data across your sharded cluster.

The field you choose as your hashed shard key should have a good cardinality, or large number of different values. Hashed keys work well with fields that increase monotonically like *ObjectId* values or timestamps.

If you shard an empty collection using a hashed shard key, MongoDB will automatically create and migrate chunks so that each shard has two chunks. You can control how many chunks MongoDB will create with the `numInitialChunks` parameter to `shardCollection` (page 876) or by manually creating chunks on the empty collection using the `split` (page 878) command.

To shard a collection using a hashed shard key, see *Shard a Collection Using a Hashed Shard Key* (page 521).

Impacts of Shard Keys on Cluster Operations

The shard key affects write and query performance by determining how the MongoDB partitions data in the cluster and how effectively the `mongos` (page 1061) instances can direct operations to the cluster. Consider the following operational impacts of shard key selection:

Write Scaling

Some possible shard keys will allow your application to take advantage of the increased write capacity that the cluster can provide, while others do not. Consider the following example where you shard by the values of the default *_id* field, which is *ObjectID*.

MongoDB generates `ObjectID` values upon document creation to produce a unique identifier for the object. However, the most significant bits of data in this value represent a time stamp, which means that they increment in a regular and predictable pattern. Even though this value has *high cardinality* (page 521), 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()` (page 974) 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 503), which is another important characteristic of shard keys.

Querying

The `mongos` (page 1061) provides an interface for applications to interact with sharded clusters that hides the complexity of *data partitioning*. A `mongos` (page 1061) receives queries from applications, and uses metadata from the *config server* (page 498), to route queries to the `mongod` (page 1049) instances with the appropriate data. While the `mongos` (page 1061) succeeds in making all querying operational in sharded environments, the *shard key* you select can have a profound affect on query performance.

See also:

The *Sharded Cluster Query Routing* (page 506) and *config server* (page 498) 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 1061) will route to a single shard, using the *shard key* and the cluster meta data from the *config server* (page 498). For queries that don't include the shard key, `mongos` (page 1061) 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 [2], the `mongos` (page 1061) 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 1061) will route queries directly to specific shards.

To select a shard key for a collection:

- determine the most commonly included fields in queries for a given application
- find which of these operations are most performance dependent.

If this field has low cardinality (i.e not sufficiently selective) you should add a second field to the shard key making a compound shard key. The data may become more splittable with a compound shard key.

See

[2] 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 *Indexing Overview* (page 309) page for more information on indexes and compound indexes.

Sharded Cluster Query Routing (page 506) for more information on query operations in the context of sharded clusters.

Sorting In sharded systems, the `mongos` (page 1061) performs a merge-sort of all sorted query results from the shards. See *Sharded Cluster Query Routing* (page 506) and *Use Indexes to Sort Query Results* (page 323) for more information.

32.3.2 Sharded Cluster High Availability

A *production* (page 500) *cluster* has no single point of failure. This section introduces the availability concerns for MongoDB deployments in general and highlights potential failure scenarios and available resolutions.

Application Servers or `mongos` Instances Become Unavailable

If each application server has its own `mongos` (page 1061) instance, other application servers can continue access the database. Furthermore, `mongos` (page 1061) instances do not maintain persistent state, and they can restart and become unavailable without loosing any state or data. When a `mongos` (page 1061) 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 367) provide high availability for shards. If the unavailable `mongod` (page 1049) is a *primary*, then the replica set will *elect* (page 389) a new primary. If the unavailable `mongod` (page 1049) is a *secondary*, and it disconnects the primary and secondary will continue to hold all data. In a three member replica set, even if a single member of the set experiences catastrophic failure, two other members have full copies of the data. [3]

Always investigate availability interruptions and failures. If a system is unrecoverable, replace it and create a new member of the replica set as soon as possible to replace the lost redundancy.

All Members of a Replica Set Become Unavailable

If all members of a replica set within a shard are unavailable, all data held in that shard is unavailable. However, the data on all other shards will remain available, and it's possible to read and write data to the other shards. However, your application must be able to deal with partial results, and you should investigate the cause of the interruption and attempt to recover the shard as soon as possible.

One or Two Config Databases Become Unavailable

Three distinct `mongod` (page 1049) instances provide the *config database* using a special two-phase commits to maintain consistent state between these `mongod` (page 1049) instances. Cluster operation will continue as normal but *chunk migration* (page 510) and the cluster can create no new *chunk splits* (page 536). 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*.

[3] If an unavailable secondary becomes available while it still has current oplog entries, it can catch up to the latest state of the set using the normal *replication process*, otherwise it must perform an *initial sync*.

Shard Keys and Cluster Availability

The most important consideration when choosing a *shard key* are:

- to ensure that MongoDB will be able to distribute data evenly among shards, and
- to scale writes across the cluster, and
- to ensure that `mongos` (page 1061) can isolate most queries to a specific `mongod` (page 1049).

Furthermore:

- Each shard should be a *replica set*, if a specific `mongod` (page 1049) 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 1061) 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.

32.3.3 Sharded Cluster Security

In most respects security for sharded clusters similar to other MongoDB deployments. Sharded clusters use the same use *keyfile* (page 212) and *access control* (page 211) as all MongoDB deployments. However, there are additional considerations when using authentication with sharded clusters.

Important: In addition to the mechanisms described in this section, always run sharded clusters in a trusted networking environment. Ensure that the network only permits trusted traffic to reach `mongos` (page 1061) and `mongod` (page 1049) instances.

See also:

Enable Authentication in a Sharded Cluster (page 522).

Access Control Privileges in Sharded Clusters

In sharded clusters, MongoDB provides separate administrative privileges for the sharded cluster and for each shard.

- Administrative privileges for the sharded cluster. These privileges give read and write access to the config servers' *admin database*. Users with these privileges can run all administrative commands. These privileges also give read and write access to all the cluster's databases. The credentials for these privileges reside on the config servers.

  To receive administrative privileges for the cluster, you must authenticate while connected to a `mongos` (page 1061) instance.

- Administrative privileges for the `mongod` (page 1049) instance, or *replica set*, that make up each individual shard. Each shard has its own `admin` database that stores the administrative credentials and access *for that shard only*. These credentials are *completely* distinct from the cluster-wide credentials.

Beyond these administration privileges, privileges for sharded clusters are functionally the same as any other MongoDB deployment. See *Access Control* (page 211) for more information.

Access a Sharded Cluster with Authentication

To access a sharded cluster as an authenticated user, from the command line, use the authentication options when connecting to a `mongos` (page 1061). Or, you can connect first and then authenticate with the `authenticate` command or the `db.auth()` (page 995) method.

To close an authenticated session, see the `logout` command.

Restriction on `localhost` Interface

Sharded clusters have restrictions on the use of `localhost` interface. If the host identifier for a MongoDB instance is either `localhost` or "`127.0.0.1`", then you must use "localhost" or "`127.0.0.1`" to identify *all* MongoDB instances in a deployment.This applies to the `host` argument to the `addShard` (page 874) command as well as to the *`--configdb`* (page 1064) option for the `mongos` (page 1061). If you mix localhost addresses with remote host address, sharded clusters will not function correctly.

32.3.4 Sharded Cluster Query Routing

MongoDB `mongos` (page 1061) instances route queries and write operations to *shards* in a sharded cluster. `mongos` (page 1061) provide the only interface to a sharded cluster from the perspective of applications. Applications never connect or communicate directly with the shards.

The `mongos` (page 1061) tracks what data is on which shard by caching the metadata from the *config servers* (page 498). The `mongos` (page 1061) uses the metadata to route operations from applications and clients to the `mongod` (page 1049) instances. A `mongos` (page 1061) has no *persistent* state and consumes minimal system resources.

The most common practice is to run `mongos` (page 1061) instances on the same systems as your application servers, but you can maintain `mongos` (page 1061) instances on the shards or on other dedicated resources.

Note: Changed in version 2.1.

Some aggregation operations using the `aggregate` (page 834) command (i.e. `db.collection.aggregate()` (page 945)) will cause `mongos` (page 1061) instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architecture decisions if you use the *aggregation framework* extensively in a sharded environment.

Routing Process

A `mongos` (page 1061) instance uses the following processes to route queries and return results.

How `mongos` Determines which Shards Receive a Query

A `mongos` (page 1061) instance routes a query to a *cluster* by:

1. Determining the list of *shards* that must receive the query.
2. Establishing a cursor on all targeted shards.

In some cases, when the *shard key* or a prefix of the shard key is a part of the query, the `mongos` (page 1061) can route the query to a subset of the shards. Otherwise, the `mongos` (page 1061) must direct the query to *all* shards that hold documents for that collection.

Example

Given the following shard key:

```
{ zipcode: 1, u_id: 1, c_date: 1 }
```

Depending on the distribution of chunks in the cluster, the mongos (page 1061) may be able to target the query at a subset of shards, if the query contains the following fields:

```
{ zipcode: 1 }
{ zipcode: 1, u_id: 1 }
{ zipcode: 1, u_id: 1, c_date: 1 }
```

How mongos Establishes a Cursor on Targeted Shards

When the first batch of results returns from the cursors, the mongos (page 1061) instance does one of the following:

- For query with sorted results (i.e. using cursor.sort() (page 991)) the mongos (page 1061) instance performs a merge sort of all queries.
- For a query with unsorted results, the mongos (page 1061) instance 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 1061) exhausted each cursor, one by one.

Detect Connections to mongos Instances

To detect if the MongoDB instance that your client is connected to is mongos (page 1061), use the isMaster (page 871) command. When a client connects to a mongos (page 1061), isMaster (page 871) 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 (page 1049), the returned document does not include the isdbgrid string.

Broadcast Operations and Targeted Operations

In general, operations in a sharded environment are either:

- Broadcast to all shards in the cluster that hold documents in a collection
- Targeted at a single shard or a limited group of shards, based on the shard key

For best performance, use targeted operations whenever possible. While some operations must broadcast to all shards, you can ensure MongoDB uses targeted operations whenever possible by always including the shard key.

Broadcast Operations

mongos (page 1061) instances broadcast queries to all shards [4] **unless** the mongos (page 1061) can determine which shard or shard stores this data.

[4] If a shard does not store chunks from a given collection, queries for documents in that collection are not broadcast to that shard.

Multi-update operations are always broadcast operations.

The `remove()` (page 970) operation is always a broadcast operation, unless the operation specifies the shard key in full.

Targeted Operations

All `insert()` (page 961) operations target to one shard.

All single `update()` (page 974) operations target to one shard. This includes *upsert* operations.

For queries that include the shard key, `mongos` (page 1061) can target the query at a specific shard or set of shards. This is the case only if the portion of the shard key included in the query is a *prefix* of the shard key. For example, if the shard key is:

```
{ a: 1, b: 1, c: 1 }
```

The `mongos` (page 1061) program *can* route queries that include the full shard key or either of the following shard key prefixes at a specific shard or set of shards:

```
{ a: 1 }
{ a: 1, b: 1 }
```

Depending on the distribution of data in the cluster and the selectivity of the query, `mongos` (page 1061) may still have to contact multiple shards [5] to fulfill these queries.

Sharded and Non-Sharded Data

Sharding operates on the collection level. You can shard multiple collections within a database or have multiple databases with sharding enabled. [6] However, in production deployments, some databases and collections will use sharding, while other databases and collections will only reside on a single shard.

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 1061) even for operations that do not impact the sharded data.

32.4 Sharding Mechanics

The following documents describe sharded cluster processes.

***Sharded Collection Balancing* (page 510)** Balancing distributes a sharded collection's data cluster to all of the shards.

***Chunk Migration Across Shards* (page 511)** MongoDB migrates chunks to shards as part of the balancing process.

***Chunk Splits in a Sharded Cluster* (page 512)** When a chunk grows beyond the configured size, MongoDB splits the chunk in half.

***Shard Key Indexes* (page 513)** Sharded collections must keep an index that starts with the shard key.

***Sharded Cluster Metadata* (page 513)** The cluster maintains internal metadata that reflects the location of data within the cluster.

[5] `mongos` (page 1061) will route some queries, even some that include the shard key, to all shards, if needed.

[6] As you configure sharding, you will use the `enableSharding` (page 875) command to enable sharding for a database. This simply makes it possible to use the `shardCollection` (page 876) command on a collection within that database.

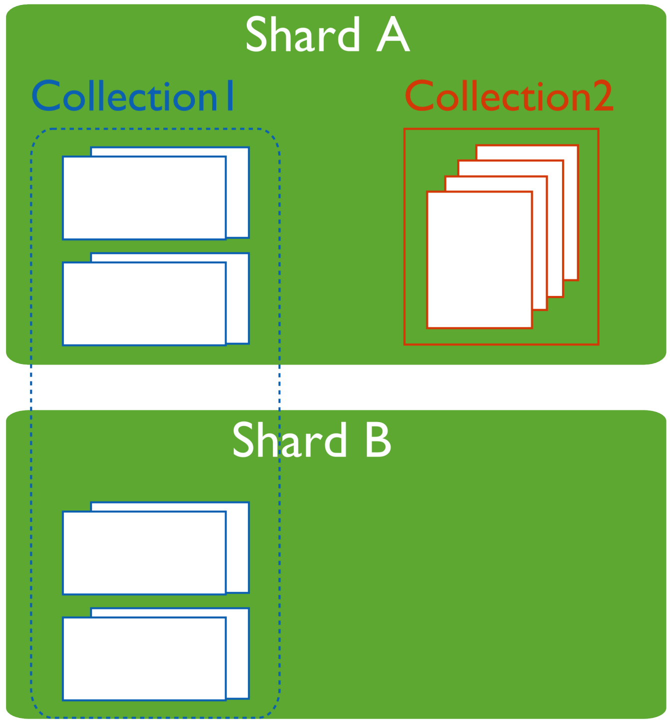


Figure 32.6: Diagram of a primary shard. A primary shard contains non-sharded collections as well as chunks of documents from sharded collections. Shard A is the primary shard.

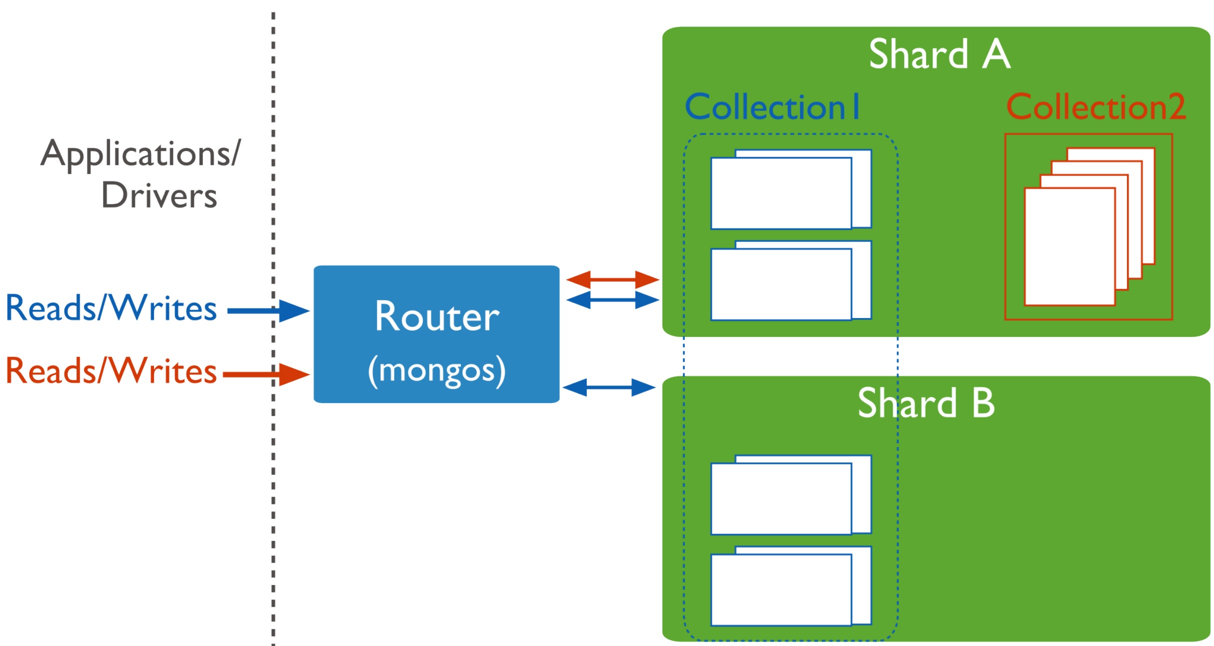


Figure 32.7: Diagram of applications/drivers issuing queries to mongos for unsharded collection as well as sharded collection. Config servers not shown.

32.4.1 Sharded Collection Balancing

Balancing is the process MongoDB uses to distribute data of a sharded collection evenly across a *sharded cluster*. When a *shard* has too many of a sharded collection's *chunks* compared to other shards, MongoDB automatically balances the the chunks across the shards. The balancing procedure for *sharded clusters* is entirely transparent to the user and application layer.

Cluster Balancer

The *balancer* process is responsible for redistributing the chunks of a sharded collection evenly among the shards for every sharded collection. By default, the balancer process is always running.

Any `mongos` (page 1061) instance in the cluster can start a balancing round. When a balancer process is active, the responsible `mongos` (page 1061) acquires a "lock" by modifying a document in the `lock` collection in the *Config Database* (page 555).

Note: Changed in version 2.0: Before MongoDB version 2.0, large differences in timekeeping (i.e. clock skew) between `mongos` (page 1061) instances could lead to failed distributed locks. This carries the possibility of data loss, particularly with skews larger than 5 minutes. Always use the network time protocol (NTP) by running `ntpd` on your servers to minimize clock skew.

To address uneven chunk distribution for a sharded collection, the balancer *migrates chunks* (page 511) from shards with more chunks to shards with a fewer number of chunks. The balancer migrates the chunks, one at a time, until there is an even dispersion of chunks for the collection across the shards.

Chunk migrations carry some overhead in terms of bandwidth and workload, both of which can impact database performance. The *balancer* attempts to minimize the impact by:

- Moving only one chunk at a time.
- Starting a balancing round **only** when the difference in the number of chunks between the shard with the greatest number of chunks for a sharded collection and the shard with the lowest number of chunks for that collection reaches the *migration threshold* (page 510).

You may disable the balancer temporarily for maintenance. See *Disable the Balancer* (page 543) for details.

You can also limit the window during which the balancer runs to prevent it from impacting production traffic. See *Schedule the Balancing Window* (page 542) for details.

Note: The specification of the balancing window is relative to the local time zone of all individual `mongos` (page 1061) instances in the cluster.

See also:

Manage Sharded Cluster Balancer (page 541).

Migration Thresholds

To minimize the impact of balancing on the cluster, the *balancer* will not begin balancing until the distribution of chunks for a sharded collection has reached certain thresholds. The thresholds apply to the difference in number of *chunks* between the shard with the most chunks for the collection and the shard with the fewest chunks for that collection. The balancer has the following thresholds:

Changed in version 2.2: The following thresholds appear first in 2.2. Prior to this release, a balancing round would only start if the shard with the most chunks had 8 more chunks than the shard with the least number of chunks.

Number of Chunks	Migration Threshold
Less than 20	2
21-80	4
Greater than 80	8

Once a balancing round starts, the balancer will not stop until, for the collection, the difference between the number of chunks on any two shards for that collection is *less than two* or a chunk migration fails.

Shard Size

By default, MongoDB will attempt to fill all available disk space with data on every shard as the data set grows. To ensure that the cluster always has the capacity to handle data growth, monitor disk usage as well as other performance metrics.

When adding a shard, you may set a "maximum size" for that shard. This prevents the *balancer* from migrating chunks to the shard when the value of `mapped` (page 924) exceeds the "maximum size". Use the `maxSize` parameter of the `addShard` (page 874) command to set the "maximum size" for the shard.

See also:

Change the Maximum Storage Size for a Given Shard (page 541) and *Monitoring for MongoDB* (page 158).

32.4.2 Chunk Migration Across Shards

Chunk migration moves the chunks of a sharded collection from one shard to another and is part of the *balancer* (page 510) process.

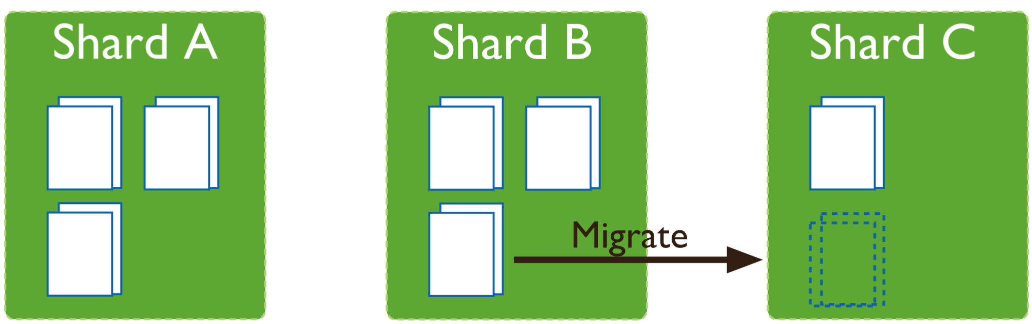


Figure 32.8: Diagram of a collection distributed across three shards. For this collection, the difference in the number of chunks between the shards reaches the *migration thresholds* (page 510) (in this case, 2) and triggers migration.

Chunk Migration

MongoDB migrates chunks in a *sharded cluster* to distribute the chunks of a sharded collection evenly among shards. Migrations may be either:

- Manual. Only use manual migration in limited cases, such as to distribute data during bulk inserts. See *Migrating Chunks Manually* (page 538) for more details.
- Automatic. The *balancer* (page 510) process automatically migrates chunks when there is an uneven distribution of a sharded collection's chunks across the shards. See *Migration Thresholds* (page 510) for more details.

All chunk migrations use the following procedure:

1. The balancer process sends the `moveChunk` (page 880) command to the source shard.
2. The source starts the move with an internal `moveChunk` (page 880) command. During the migration process, operations to the chunk route to the source shard. The source shard is responsible for incoming write operations for the chunk.
3. The destination shard begins requesting documents in the chunk and starts receiving copies of the data.
4. After receiving the final document in the chunk, the destination shard starts a synchronization process to ensure that it has the changes to the migrated documents that occurred during the migration.
5. When fully synchronized, the destination shard connects to the *config database* and updates the cluster metadata with the new location for the chunk.
6. After the destination shard completes the update of the metadata, and once there are no open cursors on the chunk, the source shard deletes its copy of the documents.

The migration process ensures consistency and maximizes the availability of chunks during balancing.

Changed in version 2.4: While copying and deleting data during migrations, the balancer waits for replication to secondaries. See Secondary Throttle in the v2.2 Manual for details.

32.4.3 Chunk Splits in a Sharded Cluster

When a chunk grows beyond the *specified chunk size* (page 512), a `mongos` (page 1061) instance will split the chunk in half. Splits may lead to an uneven distribution of the chunks for a collection across the shards. In such cases, the `mongos` (page 1061) instances will initiate a round of migrations to redistribute chunks across shards. See *Sharded Collection Balancing* (page 510) for more details on balancing chunks across shards.

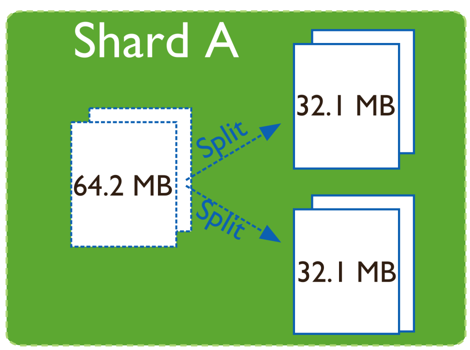


Figure 32.9: Diagram of a shard with a chunk that exceeds the default *chunk size* (page 512) of 64 MB and triggers a split of the chunk into two chunks.

Chunk Size

The default *chunk* size in MongoDB is 64 megabytes. You can *increase or reduce the chunk size* (page 538), mindful of its effect on the cluster's efficiency.

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations. This creates expense at the query routing (`mongos` (page 1061)) layer.

2. Large chunks lead to fewer migrations. This is more efficient both from the networking perspective *and* in terms of internal overhead at the query routing layer. But, these efficiencies come at the expense of a potentially more uneven distribution of data.

For many deployments, it makes sense to avoid frequent and potentially spurious migrations at the expense of a slightly less evenly distributed data set.

Limitations

Changing the chunk size affects when chunks split but there are some limitations to its effects.

- Automatic splitting only occurs during inserts or updates. If you lower the chunk size, it may take time for all chunks to split to the new size.
- Splits cannot be "undone". If you increase the chunk size, existing chunks must grow through inserts or updates until they reach the new size.

Note: Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary.

32.4.4 Shard Key Indexes

All sharded collections **must** have an index that starts with the *shard key*. If you shard a collection without any documents and *without* such an index, the `shardCollection` (page 876) command will create the index on the shard key. If the collection already has documents, you must create the index before using `shardCollection` (page 876).

Changed in version 2.2: The index on the shard key no longer needs to be only on the shard key. This index can be an index of the shard key itself, or a *compound index* where the shard key is the prefix of the index.

Important: The index on the shard key **cannot** be a *multikey index* (page 312).

A sharded collection named `people` has for its shard key the field `zipcode`. It currently has the index `{ zipcode: 1 }`. You can replace this index with a compound index `{ zipcode: 1, username: 1 }`, as follows:

1. Create an index on `{ zipcode: 1, username: 1 }`:

```
db.people.ensureIndex( { zipcode: 1, username: 1 } );
```

2. When MongoDB finishes building the index, you can safely drop the existing index on `{ zipcode: 1 }`:

```
db.people.dropIndex( { zipcode: 1 } );
```

Since the index on the shard key cannot be a multikey index, the index `{ zipcode: 1, username: 1 }` can only replace the index `{ zipcode: 1 }` if there are no array values for the `username` field.

If you drop the last valid index for the shard key, recover by recreating an index on just the shard key.

32.4.5 Sharded Cluster Metadata

Config servers (page 498) store the metadata for a sharded cluster. The metadata reflects state and organization of the sharded data sets and system. The metadata includes the list of chunks on every shard and the ranges that define the chunks. The `mongos` (page 1061) instances cache this data and use it to route read and write operations to shards.

Config servers store the metadata in the *Config Database* (page 555).

Important: Always back up the `config` database before doing any maintenance on the config server.

To access the `config` database, issue the following command from the `mongo` (page 1066) shell:

```
use config
```

In general, you should *never* edit the content of the `config` database directly. The `config` database contains the following collections:

- `changelog` (page 556)
- `chunks` (page 557)
- `collections` (page 558)
- `databases` (page 558)
- `lockpings` (page 559)
- `locks` (page 559)
- `mongos` (page 559)
- `settings` (page 560)
- `shards` (page 560)
- `version` (page 561)

For more information on these collections and their role in sharded clusters, see *Config Database* (page 555). See *Read and Write Operations on Config Servers* (page 498) for more information about reads and updates to the metadata.

CHAPTER 33

Sharded Cluster Tutorials

The following tutorials provide instructions for administering *sharded clusters*. For a higher-level overview, see *Sharding* (page 485).

33.1 Sharded Cluster Deployment Tutorials

The following tutorials provide information on deploying sharded clusters.

Deploy a Sharded Cluster **(page 516)** Set up a sharded cluster by creating the needed data directories, starting the required MongoDB instances, and configuring the cluster settings.

Considerations for Selecting Shard Keys **(page 519)** Choose the field that MongoDB uses to parse a collection's documents for distribution over the cluster's shards. Each shard holds documents with values within a certain range.

Shard a Collection Using a Hashed Shard Key **(page 521)** Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.

Enable Authentication in a Sharded Cluster **(page 522)** Control access to a sharded cluster through a key file and the keyFile setting on each of the cluster's components.

Add Shards to a Cluster **(page 523)** Add a shard to add capacity to a sharded cluster.

Convert a Replica Set to a Replicated Sharded Cluster **(page 524)** Convert a replica set to a sharded cluster in which each shard is its own replica set.

33.1.1 Deploy a Sharded Cluster

To set up a sharded cluster, complete the following sequence of tasks in the order defined below:

1. *Start the Config Server Database Instances* (page 516)
2. *Start the mongos Instances* (page 517)
3. *Add Shards to the Cluster* (page 517)
4. *Enable Sharding for a Database* (page 518)
5. *Enable Sharding for a Collection* (page 518)

> **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` (page 874) or the value to the *--configdb* run time option, then you must use "localhost" or `127.0.0.1` for *all* host settings for any MongoDB instances in the cluster. If you mix localhost addresses and remote host address, MongoDB will error.

Start the Config Server Database Instances

The config server processes are `mongod` (page 1049) instances that store the cluster's metadata. You designate a `mongod` (page 1049) as a config server using the *--configsvr* option. Each config server stores a complete copy of the cluster's metadata.

In production deployments, you must deploy exactly three config server instances, each running on different servers to assure good uptime and data safety. In test environments, you can run all three instances on a single server.

Config server instances receive relatively little traffic and demand only a small portion of system resources. Therefore, you can run an instance on a system that runs other cluster components.

1. Create data directories for each of the three config server instances. By default, a config server stores its data files in the */data/configdb* directory. You can choose a different location. To create a data directory, issue a command similar to the following:

   ```
   mkdir /data/configdb
   ```

2. Start the three config server instances. Start each by issuing a command using the following syntax:

   ```
   mongod --configsvr --dbpath <path> --port <port>
   ```

The default port for config servers is `27019`. You can specify a different port. The following example starts a config server using the default port and default data directory:

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

For additional command options, see *mongod* (page 1049) or *Configuration File Options* (page 1115).

Note: All config servers must be running and available when you first initiate a *sharded cluster*.

Start the `mongos` Instances

The `mongos` (page 1061) instances are lightweight and do not require data directories. You can run a `mongos` (page 1061) instance on a system that runs other cluster components, such as on an application server or a server running a `mongod` (page 1049) process. By default, a `mongos` (page 1061) instance runs on port `27017`.

When you start the `mongos` (page 1061) instance, specify the hostnames of the three config servers, either in the configuration file or as command line parameters. For operational flexibility, use DNS names for the config servers rather than explicit IP addresses. If you're not using resolvable hostname, you cannot change the config server names or IP addresses without a restarting *every* `mongos` (page 1061) and `mongod` (page 1049) instance.

To start a `mongos` (page 1061) instance, issue a command using the following syntax:

```
mongos --configdb <config server hostnames>
```

For example, to start a `mongos` (page 1061) that connects to config server instance running on the following hosts and on the default ports:

- `cfg0.example.net`
- `cfg1.example.net`
- `cfg2.example.net`

You would issue the following command:

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

Add Shards to the Cluster

A *shard* can be a standalone `mongod` (page 1049) or a *replica set*. In a production environment, each shard should be a replica set.

1. From a `mongo` (page 1066) shell, connect to the `mongos` (page 1061) instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

   For example, if a `mongos` (page 1061) is accessible at `mongos0.example.net` on port `27017`, issue the following command:

   ```
   mongo --host mongos0.example.net --port 27017
   ```

2. Add each shard to the cluster using the `sh.addShard()` (page 1022) method, as shown in the examples below. Issue `sh.addShard()` (page 1022) separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

You can instead use the addShard (page 874) database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see addShard (page 874).

The following are examples of adding a shard with sh.addShard() (page 1022):

- To add a shard for a replica set named rs1 with a member running on port 27017 on mongodb0.example.net, issue the following command:

  ```
  sh.addShard( "rs1/mongodb0.example.net:27017" )
  ```

  Changed in version 2.0.3.

  For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

  ```
  sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
  ```

- To add a shard for a standalone mongod (page 1049) on port 27017 of mongodb0.example.net, issue the following command:

  ```
  sh.addShard( "mongodb0.example.net:27017" )
  ```

Note: It might take some time for *chunks* to migrate to the new shard.

Enable Sharding for a Database

Before you can shard a collection, you must enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but make it possible to shard the collections in that database.

Once you enable sharding for a database, MongoDB assigns a *primary shard* for that database where MongoDB stores all data before sharding begins.

1. From a mongo (page 1066) shell, connect to the mongos (page 1061) instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

2. Issue the sh.enableSharding() (page 1024) method, specifying the name of the database for which to enable sharding. Use the following syntax:

   ```
   sh.enableSharding("<database>")
   ```

Optionally, you can enable sharding for a database using the enableSharding (page 875) command, which uses the following syntax:

```
db.runCommand( { enableSharding: <database> } )
```

Enable Sharding for a Collection

You enable sharding on a per-collection basis.

1. Determine what you will use for the *shard key*. Your selection of the shard key affects the efficiency of sharding. See the selection considerations listed in the *Considerations for Selecting Shard Key* (page 520).

2. If the collection already contains data you must create an index on the *shard key* using ensureIndex() (page 949). If the collection is empty then MongoDB will create the index as part of the sh.shardCollection() (page 1027) step.

3. Enable sharding for a collection by issuing the `sh.shardCollection()` (page 1027) method in the `mongo` (page 1066) shell. The method uses the following syntax:

```
sh.shardCollection("<database>.<collection>", shard-key-pattern)
```

Replace the `<database>.<collection>` string with the full namespace of your database, which consists of the name of your database, a dot (e.g. `.`), and the full name of the collection. The `shard-key-pattern` represents your shard key, which you specify in the same form as you would an `index` (page 949) key pattern.

Example

The following sequence of commands shards four collections:

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )

db.alerts.ensureIndex( { _id : "hashed" } )
sh.shardCollection("events.alerts", { "_id": "hashed" } )
```

In order, these operations shard:

(a) The `people` collection in the `records` database using the shard key `{ "zipcode": 1, "name": 1 }`.

This shard key distributes documents by the value of the `zipcode` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 521) by the values of the `name` field.

(b) The `addresses` collection in the `people` database using the shard key `{ "state": 1, "_id": 1 }`.

This shard key distributes documents by the value of the `state` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 521) by the values of the `_id` field.

(c) The `chairs` collection in the `assets` database using the shard key `{ "type": 1, "_id": 1 }`.

This shard key distributes documents by the value of the `type` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 521) by the values of the `_id` field.

(d) The `alerts` collection in the `events` database using the shard key `{ "_id": "hashed" }`.

New in version 2.4.

This shard key distributes documents by a hash of the value of the `_id` field. MongoDB computes the hash of the `_id` field for the *hashed index* (page 332), which should provide an even distribution of documents across a cluster.

33.1.2 Considerations for Selecting Shard Keys

Choosing a Shard Key

For many data sets, there may be no single, naturally occurring key in your collection that possesses all of the qualities of a good shard key. For these cases, you may select one of the following strategies:

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.
4. New in version 2.4: Use a *hashed shard key*. With a hashed shard key, you can choose a field that has high cardinality and create a *hashed indexes* (page 332) index on that field. MongoDB then uses the values of this hashed index as the shard key values, thus ensuring an even distribution across the shards.

From a decision making stand point, begin by finding the field that will provide the required *query isolation* (page 503), ensure that *writes will scale across the cluster* (page 503), and then add an additional field to provide additional *cardinality* (page 521) if your primary key does not have sufficient split-ability.

Considerations for Selecting Shard Key

Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster. Appropriate shard key choice depends on the schema of your data and the way that your application queries and writes data to the database.

Create a Shard Key that is Easily Divisible

An easily divisible shard key makes it easy for MongoDB to distribute content among the shards. Shard keys that have a limited number of possible values can result in chunks that are "unsplittable." See the *Cardinality* (page 521) section for more information.

Create a Shard Key that has High Randomness

A shard key with high randomness prevents any single shard from becoming a bottleneck and will distribute write operations among the cluster.

Conversely, a shard keys that has a high correlation with insert time is a poor choice. For more information, see the *Write Scaling* (page 503).

Create a Shard Key that Targets a Single Shard

A shard key that targets a single shard makes it possible for the **mongos** program to return most query operations directly from a single *specific* **mongod** instance. Your shard key should be the primary field used by your queries. Fields with a high degree of "randomness" are poor choices for this reason. For examples, see *Query Isolation* (page 503).

Shard Using a Compound Shard Key

The challenge when selecting a shard key is that there is not always an obvious choice. Often, an existing field in your collection may not be the optimal key. In those situations, computing a special purpose shard key into an additional field or using a compound shard key may help produce one that is more ideal.

Cardinality

Cardinality in the context of MongoDB, refers to the ability of the system to *partition* data into *chunks*. For example, consider a collection of data such as an "address book" that stores address records:

- Consider the use of a `state` field as a shard key:

  The state key's value holds the US state for a given address document. This field has a *low cardinality* as all documents that have the same value in the `state` field *must* reside on the same shard, even if a particular state's chunk exceeds the maximum chunk size.

  Since there are a limited number of possible values for the `state` field, MongoDB may distribute data unevenly among a small number of fixed chunks. This may have a number of effects:

  – If MongoDB cannot split a chunk because all of its documents have the same shard key, migrations involving these un-splittable chunks will take longer than other migrations, and it will be more difficult for your data to stay balanced.

  – If you have a fixed maximum number of chunks, you will never be able to use more than that number of shards for this collection.

- Consider the use of a `zipcode` field as a shard key:

  While this field has a large number of possible values, and thus has potentially higher cardinality, it's possible that a large number of users could have the same value for the shard key, which would make this chunk of users un-splittable.

  In these cases, cardinality depends on the data. If your address book stores records for a geographically distributed contact list (e.g. "Dry cleaning businesses in America,") then a value like zipcode would be sufficient. However, if your address book is more geographically concentrated (e.g "ice cream stores in Boston Massachusetts,") then you may have a much lower cardinality.

- Consider the use of a `phone-number` field as a shard key:

  Phone number has a *high cardinality,* because users will generally have a unique value for this field, MongoDB will be able to split as many chunks as needed.

While "high cardinality," is necessary for ensuring an even distribution of data, having a high cardinality does not guarantee sufficient *query isolation* (page 503) or appropriate *write scaling* (page 503).

33.1.3 Shard a Collection Using a Hashed Shard Key

New in version 2.4.

Hashed shard keys (page 502) use a *hashed index* (page 332) of a field as the *shard key* to partition data across your sharded cluster.

For tips on choosing the right field as your hashed shard key, see *Hashed Shard Keys* (page 502). For limitations on hashed indexes, see *Create a Hashed Index* (page 332).

Basic Procedure

To shard a collection using a hashed shard key, issue an operation in the `mongo` (page 1066) shell that resembles the following:

```
sh.shardCollection( "records.active", { a: "hashed" } )
```

This operation shards the `active` collection in the `records` database, using a hash of the `a` field as the shard key.

Specify the Initial Number of Chunks

If you shard an empty collection using a hashed shard key, MongoDB automatically creates and migrate chunks so that each shard has two chunks. You can control how many chunks MongoDB creates sharding the collection using `shardCollection` (page 876) with the `numInitialChunks` parameter.

Warning: MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be consistently converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

Important: MongoDB 2.4 add support for hashed shard keys. After sharding a collection with a hashed shard key, you must use the MongoDB 2.4 or higher `mongos` (page 1061) and `mongod` (page 1049) instances in your sharded cluster.

33.1.4 Enable Authentication in a Sharded Cluster

New in version 2.0: Support for authentication with sharded clusters.

To control access to a sharded cluster, create key files and then set the `keyFile` (page 1117) option on *all* components of the sharded cluster, including all `mongos` (page 1061) instances, all config server `mongod` (page 1049) instances, and all shard `mongod` (page 1049) instances. The content of the key file is arbitrary but must be the same on all cluster members.

Note: For an overview of authentication, see *Access Control* (page 211). For an overview of security, see *Security* (page 205).

Procedure

To enable authentication, do the following:

1. Generate a key file to store authentication information, as described in the *Generate a Key File* (page 227) section.
2. On each component in the sharded cluster, enable authentication by doing one of the following:
   - In the configuration file, set the `keyFile` (page 1117) option to the key file's path and then start the component, as in the following example:

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

   - When starting the component, set `--keyFile` option, which is an option for both `mongos` (page 1061) instances and `mongod` (page 1049) instances. Set the `--keyFile` to the key file's path.

   Note: The `keyFile` (page 1117) setting implies `auth` (page 1118), which means in most cases you do not need to set `auth` (page 1118) explicitly.

3. Add the first administrative user and then add subsequent users. See *Create a User Administrator* (page 224).

33.1.5 Add Shards to a Cluster

You add shards to a *sharded cluster* after you create the cluster or anytime that you need to add capacity to the cluster. If you have not created a sharded cluster, see *Deploy a Sharded Cluster* (page 516).

When adding a shard to a cluster, you should always ensure that the cluster has enough capacity to support the migration without affecting legitimate production traffic.

In production environments, all shards should be *replica sets*.

Add a Shard to a Cluster

You interact with a sharded cluster by connecting to a `mongos` (page 1061) instance.

1. From a `mongo` (page 1066) shell, connect to the `mongos` (page 1061) instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

   For example, if a `mongos` (page 1061) is accessible at `mongos0.example.net` on port `27017`, issue the following command:

   ```
   mongo --host mongos0.example.net --port 27017
   ```

2. Add each shard to the cluster using the `sh.addShard()` (page 1022) method, as shown in the examples below. Issue `sh.addShard()` (page 1022) separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the `addShard` (page 874) database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see `addShard` (page 874).

   The following are examples of adding a shard with `sh.addShard()` (page 1022):

   - To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017" )
     ```

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
     ```

   - To add a shard for a standalone `mongod` (page 1049) on port `27017` of `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "mongodb0.example.net:27017" )
     ```

   Note: It might take some time for *chunks* to migrate to the new shard.

33.1.6 Convert a Replica Set to a Replicated Sharded Cluster

Overview

Following this tutorial, you will convert a single 3-member replica set to a cluster that consists of 2 shards. Each shard will consist of an independent 3-member replica set.

The tutorial uses a test environment running on a local system UNIX-like system. You should feel encouraged to "follow along at home." If you need to perform this process in a production environment, notes throughout the document indicate procedural differences.

The procedure, from a high level, is as follows:

1. Create or select a 3-member replica set and insert some data into a collection.
2. Start the config databases and create a cluster with a single shard.
3. Create a second replica set with three new `mongod` (page 1049) instances.
4. Add the second replica set as a shard in the cluster.
5. Enable sharding on the desired collection or collections.

Process

Install MongoDB according to the instructions in the *MongoDB Installation Tutorial* (page 3).

Deploy a Replica Set with Test Data

If have an existing MongoDB *replica set* deployment, you can omit the this step and continue from *Deploy Sharding Infrastructure* (page 525).

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/manualdata/example/firstset1`
   - `http://docs.mongodb.org/manualdata/example/firstset2`
   - `http://docs.mongodb.org/manualdata/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` (page 1049) instances by running each of the following commands:

   ```
   mongod --dbpath /data/example/firstset1 --port 10001 --replSet firstset --oplogSize 700 --rest
   mongod --dbpath /data/example/firstset2 --port 10002 --replSet firstset --oplogSize 700 --rest
   mongod --dbpath /data/example/firstset3 --port 10003 --replSet firstset --oplogSize 700 --rest
   ```

   Note: The *--oplogSize 700* option restricts the size of the operation log (i.e. oplog) for each `mongod` (page 1049) instance to 700MB. Without the *--oplogSize* option, each `mongod` (page 1049) 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 (page 1066) 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 (page 1049) instances on multiple systems, replace "localhost" with a resolvable domain, hostname, or the IP address of your system.

4. In the mongo (page 1066) 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 (page 1066) 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, "ad
```

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/manualdata/example/config1`
   - `http://docs.mongodb.org/manualdata/example/config2`
   - `http://docs.mongodb.org/manualdata/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 1061) instance by running the following command:

```
mongos --configdb localhost:20001,localhost:20002,localhost:20003 --port 27017 --chunkSize 1
```

Note: If you are using the collection created earlier or are just experimenting with sharding, you can use a small *--chunkSize* (1MB works well.) The default `chunkSize` (page 1126) 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 1126) options specify the *configuration databases* (e.g. `localhost:20001`, `localhost:20002`, and `localhost:2003`). The `mongos` (page 1061) instance runs on the default "MongoDB" port (i.e. `27017`), while the databases themselves are running on ports in the `30001` series. In the this example, you may omit the *--port 27017* option, as `27017` is the default port.

4. Add the first shard in `mongos` (page 1061). In a new terminal window or GNU Screen session, add the first shard, according to the following procedure:

   (a) Connect to the `mongos` (page 1061) with the following command:

   ```
   mongo localhost:27017/admin
   ```

   (b) Add the first shard to the cluster by issuing the `addShard` (page 874) 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/manualdata/example/secondset1`
   - `http://docs.mongodb.org/manualdata/example/secondset2`
   - `http://docs.mongodb.org/manualdata/example/secondset3`

2. In three new terminal windows, start three instances of `mongod` (page 1049) 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 1124) configuration. Omit this setting in production environments.

---

3. In the mongo (page 1066) shell, connect to one mongodb instance by issuing the following command:

```
mongo localhost:10004/admin
```

4. In the mongo (page 1066) 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 1061) 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 (page 875) 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 (page 875) 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 502) and *Shard Key* (page 502) 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()` (page 1013) or `db.printShardingStatus()` (page 1010).

As clients insert additional documents into this collection, `mongos` (page 1061) distributes the documents evenly between the shards.

In the `mongo` (page 1066) shell, issue the following commands to return statics against each cluster:

```
use test
db.stats()
db.printShardingStatus()
```

Example output of the `db.stats()` (page 1013) 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()` (page 1010) 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.

33.2 Sharded Cluster Maintenance Tutorials

The following tutorials provide information in maintaining sharded clusters.

33.2.1 Administer and Manage Shard Tags

In a sharded cluster, you can use tags to associate specific ranges of a *shard key* with a specific *shard* or subset of shards.

Tag a Shard

Associate tags with a particular shard using the `sh.addShardTag()` (page 1023) method when connected to a `mongos` (page 1061) instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag `NYC` to two shards, and the tags `SFO` and `NRT` to a third shard:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

You may remove tags from a particular shard using the `sh.removeShardTag()` (page 1027) method when connected to a `mongos` (page 1061) instance, as in the following example, which removes the `NRT` tag from a shard:

```
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range

To assign a tag to a range of shard keys use the `sh.addTagRange()` (page 1023) method when connected to a `mongos` (page 1061) instance. Any given shard key range may only have *one* assigned tag. You cannot overlap defined ranges, or tag the same range more than once.

Example

Given a collection named `users` in the `records` database, sharded by the `zipcode` field. The following operations assign:

- two ranges of zip codes in Manhattan and Brooklyn the `NYC` tag
- one range of zip codes in San Francisco the `SFO` tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Note: Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Remove a Tag From a Shard Key Range

The `mongod` (page 1049) does not provide a helper for removing a tag range. You may delete tag assignment from a shard key range by removing the corresponding document from the `tags` (page 560) collection of the `config` database.

Each document in the `tags` (page 560) holds the *namespace* of the sharded collection and a minimum shard key value.

Example

The following example removes the `NYC` tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

View Existing Shard Tags

The output from `sh.status()` (page 1029) lists tags associated with a shard, if any, for each shard. A shard's tags exist in the shard's document in the `shards` (page 560) collection of the `config` database. To return all shards with a specific tag, use a sequence of operations that resemble the following, which will return only those shards tagged with `NYC`:

```
use config
db.shards.find({ tags: "NYC" })
```

You can find tag ranges for all *namespaces* in the `tags` (page 560) collection of the `config` database. The output of `sh.status()` (page 1029) displays all tag ranges. To return all shard key ranges tagged with `NYC`, use the following sequence of operations:

```
use config
db.tags.find({ tags: "NYC" })
```

33.2.2 View Cluster Configuration

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the *Config Database* (page 555). A database has sharding enabled if the value of the `partitioned` field is `true`. Connect to a `mongos` (page 1061) instance with a `mongo` (page 1066) shell, and run the following operation to get a full list of databases with sharding enabled:

```
use config
db.databases.find( { "partitioned": true } )
```

Example

You can use the following sequence of commands when to return a list of all databases in the cluster:

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

If this returns the following result set:

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "animals", "partitioned" : true, "primary" : "m0.example.net:30001" }
{ "_id" : "farms", "partitioned" : false, "primary" : "m1.example2.net:27017" }
```

Then sharding is only enabled for the `animals` database.

List Shards

To list the current set of configured shards, use the `listShards` (page 875) command, as follows:

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

View Cluster Details

To view cluster details, issue `db.printShardingStatus()` (page 1010) or `sh.status()` (page 1029). Both methods return the same output.

Example

In the following example output from `sh.status()` (page 1029)

- `sharding version` displays the version number of the shard metadata.
- `shards` displays a list of the `mongod` (page 1049) instances used as shards in the cluster.
- `databases` displays all databases in the cluster, including database that do not have sharding enabled.
- The `chunks` information for the `foo` database displays how many chunks are on each shard and displays the range of each chunk.

```
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000",  "host" : "m0.example.net:30001" }
    {  "_id" : "shard0001",  "host" : "m3.example2.net:50000" }
```

```
databases:
  {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
  {  "_id" : "contacts",  "partitioned" : true,  "primary" : "shard0000" }
      foo.contacts
          shard key: { "zip" : 1 }
          chunks:
              shard0001    2
              shard0002    3
              shard0000    2
          { "zip" : { "$minKey" : 1 } } -->> { "zip" : 56000 } on : shard0001 { "t" : 2, "i" : 0 }
          { "zip" : 56000 } -->> { "zip" : 56800 } on : shard0002 { "t" : 3, "i" : 4 }
          { "zip" : 56800 } -->> { "zip" : 57088 } on : shard0002 { "t" : 4, "i" : 2 }
          { "zip" : 57088 } -->> { "zip" : 57500 } on : shard0002 { "t" : 4, "i" : 3 }
          { "zip" : 57500 } -->> { "zip" : 58140 } on : shard0001 { "t" : 4, "i" : 0 }
          { "zip" : 58140 } -->> { "zip" : 59000 } on : shard0000 { "t" : 4, "i" : 1 }
          { "zip" : 59000 } -->> { "zip" : { "$maxKey" : 1 } } on : shard0000 { "t" : 3, "i" : 3 }
  {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
```

33.2.3 Deploy Three Config Servers for Production Deployments

This procedure converts a test deployment with only one *config server* (page 498) to a production deployment with three config servers.

For redundancy, all production *sharded clusters* (page 487) should deploy three config servers on three different machines. Use a single config server only for testing deployments, never for production deployments. When you shift to production, upgrade immediately to three config servers.

To convert a test deployment with one config server to a production deployment with three config servers:

1. Shut down all existing MongoDB processes in the cluster. This includes:

   - all `mongod` (page 1049) instances or *replica sets* that provide your shards.
   - all `mongos` (page 1061) instances in your cluster.

2. Copy the entire `dbpath` (page 1118) file system tree from the existing config server to the two machines that will provide the additional config servers. These commands, issued on the system with the existing *Config Database* (page 555), `mongo-config0.example.net` may resemble the following:

   ```
   rsync -az /data/configdb mongo-config1.example.net:/data/configdb
   rsync -az /data/configdb mongo-config2.example.net:/data/configdb
   ```

3. Start all three config servers, using the same invocation that you used for the single config server.

   ```
   mongod --configsvr
   ```

4. Restart all shard `mongod` (page 1049) and `mongos` (page 1061) processes.

33.2.4 Migrate Config Servers with the Same Hostname

This procedure migrates a *config server* (page 498) in a *sharded cluster* (page 495) to a new system that uses *the same* hostname.

1. Shut down the config server that you are moving.

   This will render all config data for your cluster *read only* (page 498).

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 1118) 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.

33.2.5 Migrate Config Servers with Different Hostnames

This procedure migrates a *config server* (page 498) in a *sharded cluster* (page 495) to a new server that uses a different hostname. Use this procedure only if the config server *will not* be accessible via the same hostname. If possible, avoid changing the hostname so that you can instead use the procedure to *migrate a config server and use the same hostname* (page 533).

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 543) for more information.

2. Shut down the *config server* (page 498) you are moving.

   This will render all config data for your cluster "read only:"

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```

3. Start the config instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

4. Shut down all existing MongoDB processes. This includes:

   - all mongod (page 1049) instances or *replica sets* that provide your shards.
   - the mongod (page 1049) instances that provide your existing *config databases* (page 555).
   - all mongos (page 1061) instances in your cluster.

5. Restart all mongod (page 1049) processes that provide the shard servers.

6. Update the *--configdb* parameter (or configdb (page 1126)) for all mongos (page 1061) instances and restart all mongos (page 1061) instances.

7. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 543) section for more information on managing the balancer process.

33.2.6 Replace a Config Server

This procedure replaces an inoperable *config server* (page 498) in a *sharded cluster* (page 495). Use this procedure only to replace a config server that has become inoperable (e.g. hardware failure).

This process assumes that the hostname of the instance will not change. If you must change the hostname of the instance, use the procedure to *migrate a config server and use a new hostname* (page 534).

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 543) for more information.

2. Provision a new system, with the same hostname as the previous host.

   You will have to ensure that the new system has the same IP address and hostname as the system it's replacing *or* you will need to modify the DNS records and wait for them to propagate.

3. Shut down *one* (and only one) of the existing config servers. Copy all of this host's `dbpath` (page 1118) file system tree from the current system to the system that will provide the new config server. This command, issued on the system with the data files, may resemble the following:

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```

4. Restart the config server process that you used in the previous step to copy the data files to the new config server instance.

5. Start the new config server instance. The default invocation is:

   ```
   mongod --configsvr
   ```

6. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 543) section for more information on managing the balancer process.

Note: In the course of this procedure *never* remove a config server from the `configdb` (page 1126) parameter on any of the `mongos` (page 1061) instances. If you need to change the name of a config server, always make sure that all `mongos` (page 1061) instances have three config servers specified in the `configdb` (page 1126) setting at all times.

33.2.7 Backup Cluster Metadata

This procedure shuts down the `mongod` (page 1049) instance of a *config server* (page 498) in order to create a backup of a *sharded cluster's* (page 487) metadata. The cluster's config servers store all of the cluster's metadata, most importantly the mapping from *chunks* to *shards*.

When you perform this procedure, the cluster remains operational [1].

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 543) for more information.
2. Shut down one of the config databases.
3. Create a full copy of the data files (i.e. the path specified by the `dbpath` (page 1118) option for the config instance.)
4. Restart the original configuration server.
5. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 543) section for more information on managing the balancer process.

See also:

Backup Strategies for MongoDB Systems (page 133).

33.2.8 Manage Chunks in a Sharded Cluster

This page 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.

[1] While one of the three config servers is unavailable, the cluster cannot split any chunks nor can it migrate chunks between shards. Your application will be able to write data to the cluster. See *Config Servers* (page 498) for more information.

Split Chunks

Normally, MongoDB splits a *chunk* following inserts when a chunk exceeds the *chunk size* (page 512). The *balancer* may migrate recently split chunks to a new shard immediately if `mongos` (page 1061) predicts future insertions will benefit from the move.

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.

> **Warning:** Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

Use `sh.status()` (page 1029) to determine the current chunks ranges across the cluster.

To split chunks manually, use the `split` (page 878) command with operators: `middle` and `find`. The equivalent shell helpers are `sh.splitAt()` (page 1028) or `sh.splitFind()` (page 1028).

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()` (page 1028) 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()` (page 1028). The query in `sh.splitFind()` (page 1028) 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()` (page 1028) 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)

Pre-splitting the chunk ranges in an empty sharded collection, allows clients to insert data into an already-partitioned collection. In most situations a *sharded cluster* will create and distribute chunks automatically without user intervention. However, in a limited number of use profiles, MongoDB cannot create enough chunks or distribute data fast enough to support required throughput. For example, if:

- 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 migrate only a single chunk at a time. Chunk migrations are resource intensive and further complicated by large write volume to the migrating chunk.

> **Warning:** You can only pre-split an empty collection. When you enable sharding for a collection that contains data MongoDB automatically creates splits. Subsequent attempts to create splits manually, can lead to unpredictable chunk ranges and sizes as well as inefficient or ineffective balancing behavior.

To create and migrate chunks manually, use the following procedure:

1. Split empty chunks in your collection by manually performing `split` (page 878) 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` (page 1066) 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` (page 880) 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 *Sharded Collection Balancing* (page 510). For lower level information on balancing, see *Cluster Balancer* (page 510).

Modify Chunk Size

When you initialize a sharded cluster, [2] 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 1061) in the cluster using the `mongo` (page 1066) shell.
2. issue the following command to switch to the *Config Database* (page 555):

   ```
   use config
   ```

3. Issue the following `save()` (page 972) 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 1126) and `--chunkSize` options, passed at runtime to the `mongos` (page 1061) **do not** affect the chunk size after you have initialized the cluster. [1]

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 510), in particular the section *Chunk Migration* (page 511).

To migrate chunks, use the `moveChunk` (page 880) command.

Note: To return a list of shards, use the `listShards` (page 875) command.

[2] The first `mongos` (page 1061) that connects to a set of *config servers* initializes the sharded cluster.

Specify shard names using the addShard (page 874) command using the name argument. If you do not specify a name in the addShard (page 874) 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 (page 1066) shell connected to any mongos (page 1061) instance.

```
db.adminCommand( { moveChunk : "myapp.users",
                   find : {username : "smith"},
                   to : "mongodb-shard3.example.net" } )
```

This command moves the chunk that includes the shard key value "smith" to the *shard* named mongodb-shard3.example.net. The command will block until the migration is complete.

See *Create Chunks (Pre-Splitting)* (page 537) for an introduction to pre-splitting.

New in version 2.2: moveChunk (page 880) command has the: _secondaryThrottle parameter. When set to true, MongoDB ensures that changes to shards as part of chunk migrations replicate to *secondaries* throughout the migration operation. For more information, see *Require Replication before Chunk Migration (Secondary Throttle)* (page 541).

Changed in version 2.4: In 2.4, _secondaryThrottle is true by default.

Warning: The moveChunk (page 880) 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*. For bulk insert operations, consider the following strategies:

- 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 537).
- You can parallelize import processes by sending insert operations to more than one mongos (page 1061) instance. If the collection is empty, pre-split first, as described in *Create Chunks (Pre-Splitting)* (page 537).
- 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 preserve 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 502) and see *Shard Key Internals* (page 502) (in particular, *Choosing a Shard Key* (page 519) and *Choosing a Shard Key* (page 519)).

33.2.9 Configure Behavior of Balancer Process in Sharded Clusters

The balancer is a process that runs on *one* of the `mongos` (page 1061) instances in a cluster and ensures that *chunks* are evenly distributed throughout a sharded cluster. In most deployments, the default balancer configuration is sufficient for normal operation. However, administrators might need to modify balancer behavior depending on application or operational requirements. If you encounter a situation where you need to modify the behavior of the balancer, use the procedures described in this document.

For conceptual information about the balancer, see *Sharded Collection Balancing* (page 510) and *Cluster Balancer* (page 510).

Schedule a Window of Time for Balancing to Occur

You can schedule a window of time during which the balancer can migrate chunks, as described in the following procedures:

- *Schedule the Balancing Window* (page 542)
- *Remove a Balancing Window Schedule* (page 543).

The `mongos` (page 1061) instances user their own local timezones to when respecting balancer window.

Configure Default Chunk Size

The default chunk size for a sharded cluster is 64 megabytes. In most situations, the default size is appropriate for splitting and migrating chunks. For information on how chunk size affects deployments, see details, see *Chunk Size* (page 512).

Changing the default chunk size affects chunks that are processes during migrations and auto-splits but does not retroactively affect all chunks.

To configure default chunk size, see *Modify Chunk Size* (page 538).

Change the Maximum Storage Size for a Given Shard

The maxSize field in the shards (page 560) collection in the *config database* (page 555) sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If dataSize (page 905) is above a shard's maxSize, the balancer will not move chunks to the shard. Also, the balancer will not move chunks off an overloaded shard. This must happen manually. The maxSize value only affects the balancer's selection of destination shards.

By default, maxSize is not specified, allowing shards to consume the total amount of available space on their machines if necessary.

You can set maxSize both when adding a shard and once a shard is running.

To set maxSize when adding a shard, set the addShard (page 874) command's maxSize parameter to the maximum size in megabytes. For example, the following command run in the mongo (page 1066) shell adds a shard with a maximum size of 125 megabytes:

```
db.runCommand( { addshard : "example.net:34008", maxSize : 125 } )
```

To set maxSize on an existing shard, insert or update the maxSize field in the shards (page 560) collection in the *config database* (page 555). Set the maxSize in megabytes.

Example

Assume you have the following shard without a maxSize field:

```
{ "_id" : "shard0000", "host" : "example.net:34001" }
```

Run the following sequence of commands in the mongo (page 1066) shell to insert a maxSize of 125 megabytes:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 125 } } )
```

To later increase the maxSize setting to 250 megabytes, run the following:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 250 } } )
```

Require Replication before Chunk Migration (Secondary Throttle)

New in version 2.2.1: _secondaryThrottle became an option to the balancer and to moveChunk (page 880) in 2.2.1. _secondaryThrottle makes it possible to require the balancer wait for replication to secondaries during migrations.

Changed in version 2.4: _secondaryThrottle became the default mode for all balancer and moveChunk (page 880) operations.

See

Secondary Throttle in the v2.2 Manual for more information on configuring _secondaryThrottle.

33.2.10 Manage Sharded Cluster Balancer

This page describes provides common administrative procedures related to balancing. For an introduction to balancing, see *Sharded Collection Balancing* (page 510). For lower level information on balancing, see *Cluster Balancer* (page 510).

See also:

Configure Behavior of Balancer Process in Sharded Clusters (page 540)

Check the Balancer Lock

To see if the balancer process is active in your *cluster*, do the following:

1. Connect to any `mongos` (page 1061) in the cluster using the `mongo` (page 1066) shell.
2. Issue the following command to switch to the *Config Database* (page 555):

   ```
   use config
   ```

3. Use the following query to return the balancer lock:

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

When this command returns, you will see output like the following:

```
{   "_id" : "balancer",
"process" : "mongos0.example.net:1292810611:1804289383",
  "state" : 2,
     "ts" : ObjectId("4d0f872630c42d1978be8a2e"),
   "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)",
    "who" : "mongos0.example.net:1292810611:1804289383:Balancer:846930886",
    "why" : "doing balance round" }
```

This output confirms that:

- The balancer originates from the `mongos` (page 1061) running on the system with the hostname `mongos0.example.net`.
- The value in the `state` field indicates that a `mongos` (page 1061) has the lock. For version 2.0 and later, the value of an active lock is `2`; for earlier versions the value is `1`.

Optional

You can also use the following shell helper, which returns a boolean to report if the balancer is active:

```
sh.getBalancerState()
```

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 1061) in the cluster using the `mongo` (page 1066) shell.
2. Issue the following command to switch to the *Config Database* (page 555):

   ```
   use config
   ```

3. Use an operation modeled on the following example `update()` (page 974) 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 1061) instance in the sharded cluster. If your mongos (page 1061) instances are physically located in different time zones, use a common time zone (e.g. GMT) to ensure that the balancer window is interpreted correctly.

For instance, running the following will force the balancer to run between 11PM and 6AM local time only:

```
db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "23:00", stop : "6:
```

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 542) 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 1061) in the cluster using the mongo (page 1066) shell.
2. Issue the following operation to disable the balancer:

   ```
   sh.setBalancerState(false)
   ```

   Note: If a migration is in progress, the system will complete the in-progress migration before stopping.

3. To later re-enable the balancer, see *Enable the Balancer* (page 544).

After disabling, you can use the following operation in the mongo (page 1066) shell to determine if there are no migrations in progress:

```
use config
while( sh.isBalancerRunning() ) {
          print("waiting...");
          sleep(1000);
}
```

The sh.setBalancerState() (page 1027) helper provides a wrapper 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 1061) in the cluster using the mongo (page 1066) shell.
2. Issue the following command to switch to the *Config Database* (page 555):

   ```
   use config
   ```

3. Issue the following update to disable the balancer:

```
db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
```

4. To later re-enable the balancer, see *Enable the Balancer* (page 544).

Enable the Balancer

Use this procedure if you have disabled the balancer and are ready to re-enable it:

1. Connect to any `mongos` (page 1061) in the cluster using the `mongo` (page 1066) shell.
2. Issue one of the following operations to enable the balancer:
   - From the `mongo` (page 1066) shell, issue:

     ```
     sh.setBalancerState(false)
     ```

   - From a driver that does not have the `sh.startBalancer()` (page 1029) helper, issue the following from the `config` database:

     ```
     db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , true )
     ```

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 133), 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 542) 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 543) for the duration of the backup procedure.

If you turn the balancer off while it is in the middle of a balancing round, the shut down is not instantaneous. The balancer completes the chunk move in-progress and then ceases all further balancing rounds.

Before starting a backup operation, confirm that the balancer is not active. You can use the following command to determine if the balancer is active:

```
!sh.getBalancerState() && !sh.isBalancerRunning()
```

When the backup procedure is complete you can reactivate the balancer process.

33.2.11 Remove Shards from an Existing Sharded Cluster

To remove a *shard* you must ensure the shard's data is migrated to the remaining shards in the cluster. This procedure describes how to safely migrate data and how to remove a shard.

This procedure describes how to safely remove a *single* shard. *Do not* use this procedure to migrate an entire cluster to new hardware. To migrate an entire shard to new hardware, migrate individual shards as if they were independent replica sets.

To remove a shard, first connect to one of the cluster's `mongos` (page 1061) instances using `mongo` (page 1066) shell. Then follow the ordered sequence of tasks on this page:

1. *Ensure the Balancer Process is Active* (page 545)
2. *Determine the Name of the Shard to Remove* (page 545)
3. *Remove Chunks from the Shard* (page 545)

4. *Check the Status of the Migration* (page 545)
5. *Move Unsharded Data* (page 546)
6. *Finalize the Migration* (page 546)

Ensure the Balancer Process is Active

To successfully migrate data from a shard, the *balancer* process **must** be active. Check the balancer state using the `sh.getBalancerState()` (page 1025) helper in the `mongo` (page 1066) shell. For more information, see the section on *balancer operations* (page 543).

Determine the Name of the Shard to Remove

To determine the name of the shard, connect to a `mongos` (page 1061) instance with the `mongo` (page 1066) shell and either:

- Use the `listShards` (page 875) command, as in the following:

  ```
  db.adminCommand( { listShards: 1 } )
  ```

- Run either the `sh.status()` (page 1029) or the `db.printShardingStatus()` (page 1010) method.

The `shards._id` field lists the name of each shard.

Remove Chunks from the Shard

Run the `removeShard` (page 875) command. This begins "draining" chunks from the shard you are removing to other shards in the cluster. For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

This operation returns immediately, with the following response:

```
{ msg : "draining started successfully" , state: "started" , shard :"mongodb0" , ok : 1 }
```

Depending on your network capacity and the amount of data, this operation can take from a few minutes to several days to complete.

Check the Status of the Migration

To check the progress of the migration at any stage in the process, run `removeShard` (page 875). For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

The command returns output similar to the following:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: NumberLong(42), dbs : NumberLong
```

In the output, the `remaining` document displays the remaining number of chunks that MongoDB must migrate to other shards and the number of MongoDB databases that have "primary" status on this shard.

Continue checking the status of the *removeShard* command until the number of chunks remaining is `0`. Then proceed to the next step.

Move Unsharded Data

If the shard is the *primary shard* for one or more databases in the cluster, then the shard will have unsharded data. If the shard is not the primary shard for any databases, skip to the next task, *Finalize the Migration* (page 546).

In a cluster, a database with unsharded collections stores those collections only on a single shard. That shard becomes the primary shard for that database. (Different databases in a cluster can have different primary shards.)

Warning: Do not perform this procedure until you have finished draining the shard.

1. To determine if the shard you are removing is the primary shard for any of the cluster's databases, issue one of the following methods:

   - `sh.status()` (page 1029)
   - `db.printShardingStatus()` (page 1010)

   In the resulting document, the `databases` field lists each database and its primary shard. For example, the following `database` field shows that the `products` database uses `mongodb0` as the primary shard:

   ```
   {  "_id" : "products",  "partitioned" : true,  "primary" : "mongodb0" }
   ```

2. To move a database to another shard, use the `movePrimary` (page 881) command. For example, to migrate all remaining unsharded data from `mongodb0` to `mongodb1`, issue the following command:

   ```
   db.runCommand( { movePrimary: "products", to: "mongodb1" })
   ```

   This command does not return until MongoDB completes moving all data, which may take a long time. The response from this command will resemble the following:

   ```
   { "primary" : "mongodb1", "ok" : 1 }
   ```

Finalize the Migration

To clean up all metadata information and finalize the removal, run `removeShard` (page 875) again. For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

A success message appears at completion:

```
{ 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.

See also:

Backup and Restore Sharded Clusters (page 146)

33.3 Sharded Cluster Data Management

The following documents provide information in managing data in sharded clusters.

***Tag Aware Sharding* (page 547)** Tags associate specific ranges of a shard key with specific shards for use in managing deployment patterns.

Enforce Unique Keys for Sharded Collections **(page 548)** Ensure that a field is always unique in all collections in a sharded cluster.

Shard GridFS Data Store **(page 550)** Choose whether to shard GridFS data in a sharded collection.

33.3.1 Tag Aware Sharding

For sharded clusters, MongoDB makes it possible to associate specific ranges of a *shard key* with a specific *shard* or subset of shards. This association dictates the policy of the cluster balancer process as it balances the *chunks* around the cluster. This capability enables the following deployment patterns:

- isolating a specific subset of data on specific set of shards.
- controlling the balancing policy so that, in a geographically distributed cluster, the most relevant portions of the data set reside on the shards with the greatest proximity to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

Note: Shard key range tags are entirely distinct from *replica set member tags* (page 401).

Hash-based sharding does not support tag-aware sharding.

Behavior and Operations

Tags in a sharded cluster are pieces of metadata that dictate the policy and behavior of the cluster *balancer*. Using tags, you may associate individual shards in a cluster with one or more tags. Then, you can assign this tag string to a range of *shard key* values for a sharded collection. When migrating a chunk, the balancer will select a destination shard based on the configured tag ranges.

The balancer migrates chunks in tagged ranges to shards with those tags, if tagged shards are not balanced. [3]

Note: Because a single chunk may span different tagged shard key ranges, the balancer may migrate chunks to tagged shards that contain values that exceed the upper bound of the selected tag range.

Example

Given a sharded collection with two configured tag ranges, such that:

- *Shard key* values between `100` and `200` have tags to direct corresponding chunks to shards tagged `NYC`.
- Shard Key values between `200` and `300` have tags to direct corresponding chunks to shards tagged `SFO`.

In this cluster, the balancer will migrate a chunk with shard key values ranging between `150` and `220` to a shard tagged `NYC`, since `150` is closer to `200` than `300`.

After configuring tags on the shards and ranges of the shard key, the cluster may take some time to reach the proper distribution of data, depending on the division of chunks (i.e. splits) and the current distribution of data in the cluster. Once configured, the balancer will respect tag ranges during future *balancing rounds* (page 510).

See also:

Administer and Manage Shard Tags (page 530)

[3] To migrate chunks in a tagged environment, the balancer selects a target shard with a tag range that has an *upper* bound that is *greater than* the migrating chunk's *lower* bound. If a shard with a matching tagged range exists, the balancer will migrate the chunk to that shard.

33.3.2 Enforce Unique Keys for Sharded Collections

Overview

The `unique` (page 949) 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 1140) because insert and indexing operations are local to each shard. [4]

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 502).

   MongoDB *can* enforce uniqueness for the *shard key*. For compound shard keys, MongoDB will enforce uniqueness on the *entire* key combination, and not for a specific component of the shard key.

   You cannot specify a unique constraint on a *hashed index* (page 315).

2. Use a secondary collection to enforce uniqueness.

   Create a minimal collection that only contains the unique field and a reference to a document in the main collection. If you always insert into a secondary collection *before* inserting to the main collection, MongoDB will produce an error if you attempt to use a duplicate key.

   Note: If you have a small data set, you may not need to shard this collection and you can create multiple unique indexes. Otherwise you can shard on a single unique key.

Always use the default *acknowledged* (page 396) *write concern* (page 395) in conjunction with a *recent MongoDB driver* (page 1223).

Unique Constraints on the Shard Key

Process

To shard a collection using the `unique` constraint, specify the `shardCollection` (page 876) command in the following form:

```
db.runCommand( { shardCollection : "test.users" , key : { email : 1 } , unique : true } );
```

Remember that the `_id` field index is always unique. By default, MongoDB inserts an `ObjectId` into the `_id` field. However, you can manually insert your own value into the `_id` field and use this as the shard key. To use the `_id` field as the shard key, use the following operation:

```
db.runCommand( { shardCollection : "test.users" } )
```

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.

[4] 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*.

- 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 503) and *query isolation* (page 503), as well as *high cardinality* (page 521). These ideal shard keys are not often the same keys that require uniqueness and requires a different approach.

Unique Constraints on Arbitrary Fields

If you cannot use a unique field as the shard key or if you need to enforce uniqueness over multiple fields, you must create another *collection* to act as a "proxy collection". This collection must contain both a reference to the original document (i.e. its `ObjectId`) and the unique key.

If you must shard this "proxy" collection, then shard on the unique key using the *above procedure* (page 548); 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 1066):

```
use records;

var primary_id = ObjectId();

db.proxy.insert({
   "_id" : primary_id
   "email" : "example@example.net"
})

// if: the above operation returns successfully,
// then continue:

db.information.insert({
   "_id" : primary_id
   "email": "example@example.net"
   // additional information...
})
```

You must insert a document into the `proxy` collection first. If this operation succeeds, the `email` field is unique, and you may continue by inserting the actual document into the `information` collection.

See

The full documentation of: `db.collection.ensureIndex()` (page 949) and `shardCollection` (page 876).

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.

33.3.3 Shard GridFS Data Store

When sharding a *GridFS* store, consider the following:

`files` Collection

Most deployments will not need to shard the `files` collection. The `files` collection is typically small, and only contains metadata. None of the required keys for GridFS lend themselves to an even distribution in a sharded situation. If you *must* shard the `files` collection, use the `_id` field possibly in combination with an application field.

Leaving `files` unsharded means that all the file metadata documents live on one shard. For production GridFS stores you *must* store the `files` collection on a replica set.

`chunks` Collection

To shard the `chunks` collection by `{ files_id : 1 , n : 1 }`, issue commands similar to the following:

```
db.fs.chunks.ensureIndex( { files_id : 1 , n : 1 } )

db.runCommand( { shardCollection : "test.fs.chunks" , key : { files_id : 1 , n : 1 } } )
```

You may also want to shard using just the `file_id` field, as in the following operation:

```
db.runCommand( { shardCollection : "test.fs.chunks" , key : {  files_id : 1 } } )
```

Important: `{ files_id : 1 , n : 1 }` and `{ files_id : 1 }` are the **only** supported shard keys for the `chunks` collection of a GridFS store.

Note: Changed in version 2.2.

Before 2.2, you had to create an additional index on `files_id` to shard using *only* this field.

The default `files_id` value is an *ObjectId*, as a result the values of `files_id` are always ascending, and applications will insert all new GridFS data to a single chunk and shard. If your write load is too high for a single server to handle, consider a different shard key or use a different value for `_id` in the `files` collection.

CHAPTER 34

Sharding Reference

34.1 Sharding Methods in the mongo Shell

Name	Description
`sh._adminCommand` (page 1021)	Runs a *database command* against the admin database, like `db.runCommand()` (page 1012), but can confirm that it is issued against a `mongos` (page 1061).
`sh._checkFullName()` (page 1021)	Tests a namespace to determine if its well formed.
`sh._checkMongos()` (page 1021)	Tests to see if the `mongo` (page 1066) shell is connected to a `mongos` (page 1061) instance.
`sh._lastMigration()` (page 1021)	Reports on the last *chunk* migration.
`sh.addShard()` (page 1022)	Adds a *shard* to a sharded cluster.
`sh.addShardTag()` (page 1023)	Associates a shard with a tag, to support *tag aware sharding* (page 547).
`sh.addTagRange()` (page 1023)	Associates range of shard keys with a shard tag, to support *tag aware sharding* (page 547).
`sh.disableBalancing()` (page 1024)	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
`sh.enableBalancing()` (page 1024)	Activates the sharded collection balancer process if previously disabled using `sh.disableBalancing()` (page 1024).
`sh.enableSharding()` (page 1024)	Enables sharding on a specific database.
`sh.getBalancerHost()` (page 1025)	Returns the name of a `mongos` (page 1061) that's responsible for the balancer process.
`sh.getBalancerState()` (page 1025)	Returns a boolean to report if the *balancer* is currently enabled.
`sh.help()` (page 1025)	Returns help text for the `sh` methods.
`sh.isBalancerRunning()` (page 1026)	Returns a boolean to report if the balancer process is currently migrating chunks.
`sh.moveChunk()` (page 1026)	Migrates a *chunk* in a *sharded cluster*.
`sh.removeShardTag()` (page 1027)	Removes the association between a shard and a shard tag shard tag.
`sh.setBalancerState()` (page 1027)	Enables or disables the *balancer* which migrates *chunks* between *shards*.
`sh.shardCollection()` (page 1027)	Enables sharding for a collection.
`sh.splitAt()` (page 1028)	Divides an existing *chunk* into two chunks using a specific value of the *shard key* as the dividing point.
`sh.splitFind()` (page 1028)	Divides an existing *chunk* that contains a document matching a query into two approximately equal chunks.
`sh.startBalancer()`	Enables the *balancer* and waits for balancing to start.

34.2 Sharding Database Commands

The following database commands support *sharded clusters*.

Name	Description
`flushRouterConfig` (page 873)	Forces an update to the cluster metadata cached by a `mongos` (page 1061).
`addShard` (page 874)	Adds a *shard* to a *sharded cluster*.
`checkShardingIndex` (page 875)	Internal command that validates index on shard key.
`enableSharding` (page 875)	Enables sharding on a specific database.
`listShards` (page 875)	Returns a list of configured shards.
`removeShard` (page 875)	Starts the process of removing a shard from a sharded cluster.
`getShardMap` (page 875)	Internal command that reports on the state of a sharded cluster.
`getShardVersion` (page 876)	Internal command that returns the *config server* version.
`setShardVersion` (page 876)	Internal command to sets the *config server* version.
`shardCollection` (page 876)	Enables the sharding functionality for a collection, allowing the collection to be sharded.
`shardingState` (page 877)	Reports whether the `mongod` (page 1049) is a member of a sharded cluster.
`unsetSharding` (page 877)	Internal command that affects connections between instances in a MongoDB deployment.
`split` (page 878)	Creates a new *chunk*.
`splitChunk` (page 879)	Internal command to split chunk. Instead use the methods `sh.splitFind()` (page 1028) and `sh.splitAt()` (page 1028).
`splitVector` (page 880)	Internal command that determines split points.
`medianKey` (page 880)	Deprecated internal command. See `splitVector` (page 880).
`moveChunk` (page 880)	Internal command that migrates chunks between shards.
`movePrimary` (page 881)	Reassigns the *primary shard* when removing a shard from a sharded cluster.
`isdbgrid` (page 881)	Verifies that a process is a `mongos` (page 1061).

34.3 Reference Documentation

***Config Database* (page 555)** Complete documentation of the content of the `local` database that MongoDB uses to store sharded cluster metadata.

***Sharding Command Quick Reference* (page 561)** A quick reference for all *commands* and `mongo` (page 1066) shell methods that support sharding and sharded clusters.

34.3.1 Config Database

The `config` database supports *sharded cluster* operation. See the *Sharding* (page 485) section of this manual for full documentation of sharded clusters.

Important: Consider the schema of the `config` database *internal* and may change between releases of MongoDB. The `config` database is not a dependable API, and users should not write data to the `config` database in the course of normal operation or maintenance.

Warning: Modification of the `config` database on a functioning system may lead to instability or inconsistent data sets. If you must modify the `config` database, use `mongodump` (page 1075) to create a full backup of the `config` database.

To access the `config` database, connect to a `mongos` (page 1061) instance in a sharded cluster, and use the following helper:

```
use config
```

You can return a list of the collections, with the following helper:

```
show collections
```

Collections

config

config.**changelog**

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `changelog` (page 556) collection stores a document for each change to the metadata of a sharded collection.

Example

The following example displays a single record of a chunk split from a `changelog` (page 556) collection:

```
{
 "_id" : "<hostname>-<timestamp>-<increment>",
 "server" : "<hostname><:port>",
 "clientAddr" : "127.0.0.1:63381",
 "time" : ISODate("2012-12-11T14:09:21.039Z"),
 "what" : "split",
 "ns" : "<database>.<collection>",
 "details" : {
    "before" : {
       "min" : {
          "<database>" : { $minKey : 1 }
       },
       "max" : {
          "<database>" : { $maxKey : 1 }
       },
       "lastmod" : Timestamp(1000, 0),
       "lastmodEpoch" : ObjectId("000000000000000000000000")
    },
    "left" : {
       "min" : {
```

```
            "<database>" : { $minKey : 1 }
         },
         "max" : {
            "<database>" : "<value>"
         },
         "lastmod" : Timestamp(1000, 1),
         "lastmodEpoch" : ObjectId(<...>)
      },
      "right" : {
         "min" : {
            "<database>" : "<value>"
         },
         "max" : {
            "<database>" : { $maxKey : 1 }
         },
         "lastmod" : Timestamp(1000, 2),
         "lastmodEpoch" : ObjectId("<...>")
      }
   }
}
```

Each document in the `changelog` (page 556) collection contains the following fields:

`config.changelog._id`
The value of `changelog._id` is: `<hostname>-<timestamp>-<increment>`.

`config.changelog.server`
The hostname of the server that holds this data.

`config.changelog.clientAddr`
A string that holds the address of the client, a `mongos` (page 1061) instance that initiates this change.

`config.changelog.time`
A *ISODate* timestamp that reflects when the change occurred.

`config.changelog.what`
Reflects the type of change recorded. Possible values are:

- `dropCollection`
- `dropCollection.start`
- `dropDatabase`
- `dropDatabase.start`
- `moveChunk.start`
- `moveChunk.commit`
- `split`
- `multi-split`

`config.changelog.ns`
Namespace where the change occurred.

`config.changelog.details`
A *document* that contains additional details regarding the change. The structure of the `details` (page 557) document depends on the type of change.

config.**chunks**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The chunks (page 557) collection stores a document for each chunk in the cluster. Consider the following example of a document for a chunk named records.pets-animal_\"cat\":

```
{
   "_id" : "mydb.foo-a_\"cat\"",
   "lastmod" : Timestamp(1000, 3),
   "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc"),
   "ns" : "mydb.foo",
   "min" : {
         "animal" : "cat"
   },
   "max" : {
         "animal" : "dog"
   },
   "shard" : "shard0004"
}
```

These documents store the range of values for the shard key that describe the chunk in the min and max fields. Additionally the shard field identifies the shard in the cluster that "owns" the chunk.

config.**collections**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The collections (page 558) collection stores a document for each sharded collection in the cluster. Given a collection named pets in the records database, a document in the collections (page 558) collection would resemble the following:

```
{
   "_id" : "records.pets",
   "lastmod" : ISODate("1970-01-16T15:00:58.107Z"),
   "dropped" : false,
   "key" : {
         "a" : 1
   },
   "unique" : false,
   "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fdc")
}
```

config.**databases**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `databases` (page 558) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` (page 558) represents each database in a distinct document. When a databases have sharding enabled, the `primary` field holds the name of the *primary shard*.

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "mydb", "partitioned" : true, "primary" : "shard0000" }
```

config.**lockpings**

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `lockpings` (page 559) collection keeps track of the active components in the sharded cluster. Given a cluster with a `mongos` (page 1061) running on `example.com:30000`, the document in the `lockpings` (page 559) collection would resemble:

```
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```

config.**locks**

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `locks` (page 559) collection stores a distributed lock. This ensures that only one `mongos` (page 1061) instance can perform administrative tasks on the cluster at once. The `mongos` (page 1061) 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 1061) holds the balancer lock, the `state` field has a value of `2`, which means that balancer is active. The `when` field indicates when the balancer began the current operation.

Changed in version 2.0: The value of the `state` field was `1` before MongoDB 2.0.

config.**mongos**

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `mongos` (page 559) collection stores a document for each `mongos` (page 1061) instance affiliated with the cluster. `mongos` (page 1061) instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the `mongos` (page 1061) is active. The `ping` field shows the time of the last ping, while the `up` field reports the uptime of the mongos (page 1061) as of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the mongos (page 1061) running on example.com:30000.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait
```

config.**settings**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The settings (page 560) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see *Modify Chunk Size* (page 538).
- Balancer status. To change status, see *Disable the Balancer* (page 543).

The following is an example settings collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }
```

config.**shards**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The shards (page 560) collection represents each shard in the cluster in a separate document. If the shard is a replica set, the host field displays the name of the replica set, then a slash, then the hostname, as in the following example:

```
{ "_id" : "shard0000", "host" : "shard1/localhost:30000" }
```

If the shard has *tags* (page 547) assigned, this document has a tags field, that holds an array of the tags, as in the following example:

```
{ "_id" : "shard0001", "host" : "localhost:30001", "tags": [ "NYC" ] }
```

config.**tags**

Internal MongoDB Metadata

The config (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The tags (page 560) collection holds documents for each tagged shard key range in the cluster. The documents in the tags (page 560) collection resemble the following:

```
{
    "_id" : { "ns" : "records.users", "min" : { "zipcode" : "10001" } },
    "ns" : "records.users",
    "min" : { "zipcode" : "10001" },
```

```
    "max" : { "zipcode" : "10281" },
    "tag" : "NYC"
}
```

`config.version`

Internal MongoDB Metadata

The `config` (page 556) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `version` (page 561) collection holds the current metadata version number. This collection contains only one document:

To access the `version` (page 561) collection you must use the `db.getCollection()` (page 1005) method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

Note: Like all databases in MongoDB, the `config` database contains a `system.indexes` (page 1133) collection contains metadata for all indexes in the database for information on indexes, see *Indexes* (page 307).

34.3.2 Sharding Command Quick Reference

JavaScript Methods

Definition

`sh.addShard(host)`

Adds a database instance or replica set to a *sharded cluster*. The optimal configuration is to deploy shards across *replica sets*. This method must be run on a `mongos` (page 1061) instance.

The `sh.addShard()` (page 1022) method has the following parameter:

param string host The hostname of either a standalone database instance or of a replica set. Include the port number if the instance is running on a non-standard port. Include the replica set name if the instance is a replica set, as explained below.

The `sh.addShard()` (page 1022) method has the following prototype form:

```
sh.addShard("<host>")
```

The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[replica-set-name]/[hostname]
[replica-set-name]/[hostname]:port
```

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The `sh.addShard()` (page 1022) method is a helper for the `addShard` (page 874) command. The `addShard` (page 874) command has additional options which are not available with this helper.

Example

To add a shard on a replica set, specify the name of the replica set and the hostname of at least one member of the replica set, as a seed. If you specify additional hostnames, all must be members of the same replica set.

The following example adds a replica set named `repl0` and specifies one member of the replica set:

```
sh.addShard("repl0/mongodb3.example.net:27327")
```

Definition

`sh.enableSharding(database)`

Enables sharding on the specified database. This does not automatically shard any collections but makes it possible to begin sharding collections using `sh.shardCollection()` (page 1027).

The `sh.enableSharding()` (page 1024) method has the following parameter:

param string database The name of the database shard. Enclose the name in quotation marks.

See also:

`sh.shardCollection()` (page 1027)

Definition

`sh.shardCollection(namespace, key, unique)`

Shards a collection using the `key` as a the *shard key*. `sh.shardCollection()` (page 1027) takes the following arguments:

param string namespace The *namespace* of the collection to shard.

param document key A *document* that specifies the *shard key* to use to *partition* and distribute objects among the shards. A shard key may be one field or multiple fields. A shard key with multiple fields is called a "compound shard key."

param Boolean unique When true, ensures that the underlying index enforces a unique constraint. *Hashed shard keys* do not support unique constraints.

New in version 2.4: Use the form `{field: "hashed"}` to create a *hashed shard key*. Hashed shard keys may not be compound indexes.

> **Warning:** MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 876). Additionally, after `shardCollection` (page 876), you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

`shardCollection` (page 876) for additional options, *Sharding* (page 485) and *Sharding Introduction* (page 487) for an overview of sharding, *Deploy a Sharded Cluster* (page 516) for a tutorial, and *Shard Keys* (page 502) for choosing a shard key.

Example

Given the `people` collection in the `records` database, the following command shards the collection by the `zipcode` field:

```
sh.shardCollection("records.people", { zipcode: 1} )
```

Definition

`sh.splitFind(namespace, query)`

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` (page 1028) 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()` (page 1028).

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Definition

`sh.splitAt(namespace, query)`

Splits the chunk containing the document specified by the query as if that document were at the "middle" of the collection, even if the specified document is not the actual median of the collection.

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Use this command to manually split chunks unevenly. Use the "`sh.splitFind()` (page 1028)" 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()` (page 1028).

Definition

`sh.moveChunk(namespace, query, destination)`

Moves the *chunk* that contains the document specified by the `query` to the `destination` shard. `sh.moveChunk()` (page 1026) provides a wrapper around the `moveChunk` (page 880) database command and takes the following arguments:

param string namespace The *namespace* of the sharded collection that contains the chunk to migrate.

param document query An equality match on the shard key that selects the chunk to move.

param string destination The name of the shard to move.

Important: In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` (page 1026) directly.

See also:

`moveChunk` (page 880), `sh.splitAt()` (page 1028), `sh.splitFind()` (page 1028), *Sharding* (page 485), and *chunk migration* (page 511).

Example

Given the `people` collection in the `records` database, the following operation finds the chunk that contains the documents with the `zipcode` field set to `53187` and then moves that chunk to the shard named `shard0019`:

```
sh.moveChunk("records.people", { zipcode: 53187 }, "shard0019")
```

Description

`sh.setBalancerState(state)`

Enables or disables the *balancer*. Use `sh.getBalancerState()` (page 1025) to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` (page 1026) to check its current state.

The `sh.getBalancerState()` (page 1025) method has the following parameter:

param Boolean state Set this to `true` to enable the balancer and `false` to disable it.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

`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()` (page 1025) to determine if the balancer is enabled or disabled.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)

- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

`sh.status()`

Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*. The default behavior suppresses the detailed chunk information if the total number of chunks is greater than or equal to 20.

The `sh.status()` (page 1029) method has the following parameter:

param Boolean verbose If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See also:

`db.printShardingStatus()` (page 1010)

Definition

`sh.addShardTag(shard, tag)`

New in version 2.2.

Associates a shard with a tag or identifier. MongoDB uses these identifiers to direct *chunks* that fall within a tagged range to specific shards. `sh.addTagRange()` (page 1023) associates chunk ranges with tag ranges.

param string shard The name of the shard to which to give a specific tag.

param string tag The name of the tag to add to the shard.

Always issue `sh.addShardTag()` (page 1023) when connected to a `mongos` (page 1061) instance.

Example

The following example adds three tags, `NYC`, `LAX`, and `NRT`, to three shards:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "LAX")
sh.addShardTag("shard0002", "NRT")
```

See also:

`sh.addTagRange()` (page 1023) and `sh.removeShardTag()` (page 1027).

Definition

`sh.addTagRange(namespace, minimum, maximum, tag)`

New in version 2.2.

Attaches a range of shard key values to a shard tag created using the `sh.addShardTag()` (page 1023) method. `sh.addTagRange()` (page 1023) takes the following arguments:

param string namespace The *namespace* of the sharded collection to tag.

param document minimum The minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param document maximum The maximum value of the shard key range to include in the tag. Specify the maximum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param string tag The name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

Use `sh.addShardTag()` (page 1023) to ensure that the balancer migrates documents that exist within the specified range to a specific shard or set of shards.

Always issue `sh.addTagRange()` (page 1023) when connected to a `mongos` (page 1061) instance.

Note: If you add a tag range to a collection using `sh.addTagRange()` (page 1023) and then later drop the collection or its database, MongoDB does not remove the tag association. If you later create a new collection with the same name, the old tag association will apply to the new collection.

Example

Given a shard key of `{state: 1, zip: 1}`, the following operation creates a tag range covering zip codes in New York State:

```
sh.addTagRange( "exampledb.collection",
                { state: "NY", zip: MinKey },
                { state: "NY", zip: MaxKey },
                "NY"
              )
```

Definition

`sh.removeShardTag(shard, tag)`

New in version 2.2.

Removes the association between a tag and a shard. Always issue `sh.removeShardTag()` (page 1027) when connected to a `mongos` (page 1061) instance.

param string shard The name of the shard from which to remove a tag.

param string tag The name of the tag to remove from the shard.

See also:

`sh.addShardTag()` (page 1023), `sh.addTagRange()` (page 1023)

`sh.help()`

Returns a basic help text for all sharding related shell functions.

Database Commands

The following database commands support *sharded clusters*.

Definition

addShard

Adds either a database instance or a *replica set* to a *sharded cluster*. The optimal configuration is to deploy shards across replica sets.

Run addShard (page 874) when connected to a mongos (page 1061) instance. The command takes the following form when adding a single database instance as a shard:

```
{ addShard: "<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

When adding a replica set as a shard, use the following form:

```
{ addShard: "<replica_set>/<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

The command contains the following fields:

field string addShard The hostname and port of the mongod (page 1049) instance to be added as a shard. To add a replica set as a shard, specify the name of the replica set and the hostname and port of a member of the replica set.

field integer maxSize The maximum size in megabytes of the shard. If you set `maxSize` to `0`, MongoDB does not limit the size of the shard.

field string name A name for the shard. If this is not specified, MongoDB automatically provides a unique name.

The addShard (page 874) command stores shard configuration information in the *config database*.

Specify a `maxSize` when you have machines with different disk capacities, or if you want to limit the amount of data on some shards. The `maxSize` constraint prevents the *balancer* from migrating chunks to the shard when the value of mem.mapped (page 924) exceeds the value of `maxSize`.

Examples

The following command adds the database instance running on port``27027`` on the host `mongodb0.example.net` as a shard:

```
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The following command adds a replica set as a shard:

```
db.runCommand( { addShard: "repl0/mongodb3.example.net:27327"} )
```

You may specify all members in the replica set. All additional hostnames must be members of the same replica set.

listShards

Use the listShards (page 875) command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

enableSharding

The enableSharding (page 875) command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: "<database name>" }
```

Once you've enabled sharding in a database, you can use the shardCollection (page 876) command to begin the process of distributing data among the shards.

Definition

shardCollection

Enables a collection for sharding and allows MongoDB to begin distributing data among shards. You must run `enableSharding` (page 875) on a database before running the `shardCollection` (page 876) command. `shardCollection` (page 876) has the following form:

```
{ shardCollection: "<database>.<collection>", key: <shardkey> }
```

`shardCollection` (page 876) has the following fields:

field string shardCollection The *namespace* of the collection to shard in the form `<database>.<collection>`.

field document key The *index specification document* (page 66) to use as the shard key. The index must exist prior to the `shardCollection` (page 876) command, unless the collection is empty. If the collection is empty, in which case MongoDB creates the index prior to sharding the collection. New in version 2.4: The key may be in the form `{ field : "hashed" }`, which will use the specified field as a hashed shard key.

field Boolean unique When `true`, the `unique` option ensures that the underlying index enforces a unique constraint. Hashed shard keys do not support unique constraints.

field integer numInitialChunks To support *hashed sharding* (page 502) added in MongoDB 2.4, `numInitialChunks` specifies the number of chunks to create when sharding an collection with a hashed shard key. MongoDB will then create and balance chunks across the cluster. The `numInitialChunks` must be less than `8192`.

Shard Keys

Choosing the best shard key to effectively distribute load among your shards requires some planning. Review *Shard Keys* (page 502) regarding choosing a shard key.

Hashed Shard Keys

New in version 2.4.

Hashed shard keys (page 502) use a hashed index of a single field as the shard key.

> **Warning:** MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 876). Additionally, after `shardCollection` (page 876), you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

Sharding (page 485), *Sharding Concepts* (page 495), and *Deploy a Sharded Cluster* (page 516).

Example

The following operation enables sharding for the `people` collection in the `records` database and uses the `zipcode` field as the *shard key* (page 502):

```
db.runCommand( { shardCollection: "records.people", key: { zipcode: 1 } } )
```

shardingState

shardingState (page 877) is an admin command that reports if mongod (page 1049) is a member of a *sharded cluster*. shardingState (page 877) has the following prototype form:

```
{ shardingState: 1 }
```

For shardingState (page 877) to detect that a mongod (page 1049) is a member of a sharded cluster, the mongod (page 1049) must satisfy the following conditions:

1. the mongod (page 1049) is a primary member of a replica set, and
2. the mongod (page 1049) instance is a member of a sharded cluster.

If shardingState (page 877) detects that a mongod (page 1049) is a member of a sharded cluster, shardingState (page 877) returns a document that resembles the following prototype:

```
{
  "enabled" : true,
  "configServer" : "<configdb-string>",
  "shardName" : "<string>",
  "shardHost" : "string:",
  "versions" : {
       "<database>.<collection>" : Timestamp(<...>),
       "<database>.<collection>" : Timestamp(<...>)
  },
  "ok" : 1
}
```

Otherwise, shardingState (page 877) will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from shardingState (page 877) when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: mongos (page 1061) instances do not provide the shardingState (page 877).

Warning: This command obtains a write lock on the affected database and will block other operations until it has completed; however, the operation is typically short lived.

removeShard

Starts the process of removing a shard from a *cluster*. This is a multi-stage process. Begin by issuing the following command:

```
{ removeShard : "[shardName]" }
```

The balancer will then migrate chunks from the shard specified by `[shardName]`. This process happens slowly to avoid placing undue load on the overall cluster.

The command returns immediately, with the following message:

```
{ msg : "draining started successfully" , state: "started" , shard: "shardName" , ok : 1 }
```

If you run the command again, you'll see the following progress output:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 23 , dbs: 1 }, ok: 1 }
```

The `remaining` *document* specifies how many chunks and databases remain on the shard. Use db.printShardingStatus() (page 1010) to list the databases that you must move from the shard.

Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one of the cluster's databases, then you must manually move the database to a new shard. This can be only after the shard is empty. See the `movePrimary` (page 881) command for details.

After removing all chunks and databases from the shard, you may issue the command again, to return:

```
{ msg: "remove shard completed successfully", state: "completed", host: "shardName", ok : 1 }
```

Part X

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 *MongoDB Drivers and Client Libraries* (page 575).

For an overview application-development specific concepts, see the *aggregation* (page 245) and *indexes* (page 307) documents. For an introduction to basic MongoDB use, see the *administration tutorials* (page 171).

See also:

Core MongoDB Operations (CRUD) (page 39) section and the *FAQ: MongoDB for Application Developers* (page 735) document. Developers should also be familiar with the **mongo** shell and the MongoDB *query and update operators* (page 785).

CHAPTER 35

Development Considerations

The following documents outline basic application development documents:

35.1 MongoDB Drivers and Client Libraries

An application communicates with MongoDB by way of a client library, called a driver, that handles all interaction with the database in a language appropriate to the application.

35.1.1 Drivers

See the following pages for more information about the MongoDB drivers:

- JavaScript (Language Center, docs)
- Python (Language Center, docs)
- Ruby (Language Center, docs)
- PHP (Language Center, docs)
- Perl (Language Center, docs)
- Java (Language Center, docs)
- Scala (Language Center, docs)
- C# (Language Center, docs)
- C (Language Center, docs)
- C++ (Language Center, docs)
- Haskell (Language Center, docs)
- Erlang (Language Center, docs)

35.1.2 Driver Version Numbers

Driver version numbers use semantic versioning or "**major.minor.patch**" versioning system. The first number is the major version, the second the minor version, and the third indicates a patch.

Example

Driver version numbers.

If your driver has a version number of `2.9.1`, `2` is the major version, `9` is minor, and `1` is the patch.

The numbering scheme for drivers differs from the scheme for the MongoDB server. For more information on server versioning, see *MongoDB Version Numbers* (page 1225).

35.2 Optimization Strategies for MongoDB

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

This section describes techniques for optimizing application performance with MongoDB.

35.2.1 Optimize Query Performance with Indexes and Projections

Create Indexes to Support Queries

For commonly issued queries, create *indexes* (page 307). If a query searches multiple fields, create a *compound index* (page 311). Scanning an index is much faster than scanning a collection. The indexes structures are smaller than the documents reference, and store references in order.

Example

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

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

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

Example

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

Creating this index:

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

Optimizes this query:

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

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

Indexes support queries, update operations, and some phases of the *aggregation pipeline* (page 249).

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

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

Limit the Number of Query Results to Reduce Network Demand

MongoDB *cursors* return results in groups of multiple documents. If you know the number of results you want, you can reduce the demand on network resources by issuing the `cursor.limit()` (page 985) method.

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

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

For more information on limiting results, see `cursor.limit()` (page 985)

Use Projections to Return Only Necessary Data

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

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

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

For more information on using projections, see *Result Projections* (page 45).

Use `$hint` to Select a Particular Index

In most cases the *query optimizer* (page 48) selects the optimal index for a specific operation; however, you can force MongoDB to use a specific index using the `hint()` (page 985) method. Use `hint()` (page 985) to support performance testing, or on some queries where you must select a field or field included in several indexes.

Use the Increment Operator to Perform Operations Server-Side

Use MongoDB's `$inc` (page 810) operator to increment or decrement values in documents. The operator increments the value of the field on the server side, as an alternative to selecting a document, making simple modifications in the client and then writing the entire document to the server. The `$inc` (page 810) operator can also help avoid race conditions, which would result when two application instances queried for a document, manually incremented a field, and saved the entire document back at the same time.

35.2.2 Evaluate Performance of Current Operations

The following sections describe techniques for evaluating operational performance.

Use the Database Profiler to Evaluate Operations Against the Database

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

For more information, see *Database Profiling* (page 163).

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

The `db.currentOp()` (page 998) method reports on current operations running on a `mongod` (page 1049) instance.

Use `$explain` to Evaluate Query Performance

The `explain()` (page 979) method returns statistics on a query, and reports the index MongoDB selected to fulfill the query, as well as information about the internal operation of the query.

Example

To use `explain()` (page 979) on a query for documents matching the expression `{ a: 1 }`, in the collection named `records`, use an operation that resembles the following in the `mongo` (page 1066) shell:

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

35.2.3 Use Capped Collections for Fast Writes and Reads

Use Capped Collections for Fast Writes

Capped Collections (page 578) are circular, fixed-size collections that keep documents well-ordered, even without the use of an index. This means that capped collections can receive very high-speed writes and sequential reads.

These collections are particularly useful for keeping log files but are not limited to that purpose. Use capped collections where appropriate.

Use Natural Order for Fast Reads

To return documents in the order they exist on disk, return sorted operations using the `$natural` (page 833) operator. On a capped collection, this also returns the documents in the order in which they were written.

Natural order does not use indexes but can be fast for operations when you want to select the first or last items on disk.

See also:

`sort()` (page 991) and `limit()` (page 985).

See also:

Server-side JavaScript (page 581).

35.3 Capped Collections

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

See `db.createCollection()` (page 997) or `createCollection` for more information on creating capped collections.

Capped collections have the following behaviors:

- Capped collections guarantee preservation of the insertion order. As a result, queries do not need an index to return documents in insertion order. Without this indexing overhead, they can support higher insertion through-put.
- Capped collections guarantee that insertion order is identical to the order on disk (*natural order*) and do so by prohibiting updates that increase document size. Capped collections only allow updates that fit the original document size, which ensures a document does not change its location on disk.

- Capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations.

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

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

35.3.1 Recommendations and Restrictions

- You *can* update documents in a collection after inserting them. *However,* these updates **cannot** cause the documents to grow. If the update operation causes the document to grow beyond their original size, the update operation will fail.

  If you plan to update documents in a capped collection, create an index so that these update operations do not require a table scan.
- You cannot delete documents from a capped collection. To remove all records from a capped collection, use the 'emptycapped' command. To remove the collection entirely, use the `drop()` (page 948) method.
- You cannot shard a capped collection.
- Capped collections created after 2.2 have an `_id` field and an index on the `_id` field by default. Capped collections created before 2.2 do not have an index on the `_id` field by default. If you are using capped collections with replication prior to 2.2, you should explicitly create an index on the `_id` field.

  Warning: If you have a capped collection in a *replica set* outside of the `local` database, before 2.2, you should create a unique index on `_id`. Ensure uniqueness using the `unique: true` option to the `ensureIndex()` (page 949) method or by using an *ObjectId* for the `_id` field. Alternately, you can use the `autoIndexId` option to `create` (page 885) when creating the capped collection, as in the *Query a Capped Collection* (page 580) procedure.

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

35.3.2 Procedures

Create a Capped Collection

You must create capped collections explicitly using the `createCollection()` (page 997) method, which is a helper in the `mongo` (page 1066) shell for the `create` (page 885) command. When creating a capped collection you must specify the maximum size of the collection in bytes, which MongoDB will pre-allocate for the collection. The size of the capped collection includes a small amount of space for internal overhead.

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

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

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

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

See

`db.createCollection()` (page 997) and `create` (page 885).

Query a Capped Collection

If you perform a `find()` (page 951) on a capped collection with no ordering specified, MongoDB guarantees that the ordering of results is the same as the insertion order.

To retrieve documents in reverse insertion order, issue `find()` (page 951) along with the `sort()` (page 991) method with the `$natural` (page 833) parameter set to `-1`, as shown in the following example:

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

Check if a Collection is Capped

Use the `db.collection.isCapped()` (page 963) method to determine if a collection is capped, as follows:

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

Convert a Collection to Capped

You can convert a non-capped collection to a capped collection with the `convertToCapped` (page 887) command:

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

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

Changed in version 2.2: Before 2.2, capped collections did not have an index on `_id` unless you specified `autoIndexId` to the `create` (page 885), after 2.2 this became the default.

Automatically Remove Data After a Specified Period of Time

For additional flexibility when expiring data, consider MongoDB's *TTL* indexes, as described in *Expire Data from Collections by Setting TTL* (page 599). These indexes allow you to expire and remove data from normal collections using a special type, based on the value of a date-typed field and a TTL value for the index.

TTL Collections (page 599) are not compatible with capped collections.

Tailable Cursor

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

See *Create Tailable Cursor* (page 590) for information on creating a tailable cursor.

35.4 Server-side JavaScript

Changed in version 2.4: The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` (page 1049) could only run a single JavaScript operation at a time.

35.4.1 Overview

MongoDB supports the execution of JavaScript code for the following server-side operations:

- `mapReduce` (page 840) and the corresponding `mongo` (page 1066) shell method `db.collection.mapReduce()` (page 963). See *Map-Reduce* (page 291) for more information.
- `eval` (page 862) command, and the corresponding `mongo` (page 1066) shell method `db.eval()` (page 1002)
- `$where` (page 797) operator
- *Running .js files via a mongo shell Instance on the Server* (page 581)

JavaScript in MongoDB

Although the above operations use JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 575) in the language of the interacting application.

See also:

Store a JavaScript Function on the Server (page 582)

You can disable all server-side execution of JavaScript, by passing the *--noscripting* option on the command line or setting `noscripting` (page 1120) in a configuration file.

35.4.2 Running `.js` files via a `mongo` shell Instance on the Server

You can run a JavaScript (`.js`) file using a `mongo` (page 1066) shell instance on the server. This is a good technique for performing batch administrative work. When you run `mongo` (page 1066) shell on the server, connecting via the localhost interface, the connection is fast with low latency.

The *command helpers* (page 619) provided in the `mongo` (page 1066) shell are not available in JavaScript files because they are not valid JavaScript. The following table maps the most common `mongo` (page 1066) shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.system.users.find()`
`show log <logname>`	`db.adminCommand( { 'getLog' : '<logname>' } )`
`show logs`	`db.adminCommand( { 'getLog' : '*' } )`

35.4.3 Concurrency

Refer to the individual method or operator documentation for any concurrency information. See also the *concurrency table* (page 750).

35.5 Store a JavaScript Function on the Server

Note: We do **not** recommend using server-side stored functions if possible.

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

To store a function, you can use the `db.collection.save()` (page 972), as in the following example:

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

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

Once you save a function in the `system.js` collection, you can use the function from any JavaScript context (e.g. `eval` (page 862) command or the `mongo` (page 1066) shell method `db.eval()` (page 1002), `$where` (page 797) operator, `mapReduce` (page 840) or `mongo` (page 1066) shell method `db.collection.mapReduce()` (page 963)).

Consider the following example from the `mongo` (page 1066) shell that first saves a function named `echoFunction` to the `system.js` collection and calls the function using `db.eval()` (page 1002) method:

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

```
db.eval( "echoFunction( 'test' )" )
```

See http://github.com/mongodb/mongo/tree/master/jstests/storefunc.js for a full example.

New in version 2.1: In the `mongo` (page 1066) shell, you can use `db.loadServerScripts()` (page 1009) to load all the scripts saved in the `system.js` collection for the current database. Once loaded, you can invoke the functions directly in the shell, as in the following example:

```
db.loadServerScripts();

echoFunction(3);

myAddFunction(3, 5);
```

See also:

- *Read Preference* (page 398)
- *Write Concern* (page 395)
- *Indexing Strategies* (page 321)
- *Aggregation Framework* (page 247)
- *Map-Reduce* (page 291)
- *Perform Incremental Map-Reduce* (page 293)
- *Troubleshoot the Map Function* (page 297)
- *Troubleshoot the Reduce Function* (page 298)
- *Connection String URI Format* (page 1142)

CHAPTER 36

Application Design Patterns for MongoDB

The following documents provide patterns for developing application features:

36.1 Perform Two Phase Commits

36.1.1 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 588) like functionality.

36.1.2 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 588).

Note: Because only single-document operations are atomic with MongoDB, two-phase commits can only offer transaction-*like* semantics. It's possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

36.1.3 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` (page 1066) 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()` (page 951):

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

`mongo` (page 1066) 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()` (page 951):

```
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()` (page 955):

```
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()` (page 974) to change the value of `state` to `pending`:

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

The `find()` (page 951) 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()` (page 974) query will prevent you from applying the transaction *if* the transaction is *not* already pending. Use the following `update()` (page 974) 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()` (page 951) 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()` (page 974) operation to set the transaction's state to `committed`:

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

The `find()` (page 951) 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()` (page 974) 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()` (page 951) 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()` (page 951) operation will return the contents of the `transactions` collection, which should now resemble the following:

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

Recovering from Failure Scenarios

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

There are two classes of failures:

- all failures that occur after the first step (i.e. "*setting the transaction set to initial* (page 586)") but before the third step (i.e. "*applying the transaction to both accounts* (page 587).")

  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 586).")

- all failures that occur after the third step (i.e. "*applying the transaction to both accounts* (page 587)") but before the fifth step (i.e. "*setting the transaction state to done* (page 587).")

  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 587).")

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 587) (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 586) (i.e. the first step,) but before you *apply the transaction* (page 587) (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()` (page 974) 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()` (page 951) operation will return the contents of the `accounts` collection, which should resemble the following:

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

Set Transaction State to Canceled Finally, use the following `update()` (page 974) 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 `A2`). If both applications begin processing the transaction which is still in the `initial` state (i.e. *step 1* (page 586)), 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()` (page 952) 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` (page 1066) 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 588), 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"
```

36.1.4 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 586) (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 587) (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()` (page 974) 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` (page 1049) instance has *journaling* enabled to ensure that your data is always in a recoverable state, in the event of an unclean `mongod` (page 1049) shutdown.

36.2 Create Tailable Cursor

36.2.1 Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for *capped collections* (page 578) you may use a *Tailable Cursor* that remains open after the client exhausts the results in the initial cursor. Tailable cursors are conceptually equivalent to the `tail` Unix command with the `-f` option (i.e. with "follow" mode.) After clients insert new additional documents into a capped collection, the tailable cursor will continue to retrieve documents.

Use tailable cursors on capped collections with high numbers of write operations for which an index would be too expensive. For instance, MongoDB *replication* (page 367) uses tailable cursors to tail the primary's *oplog*.

Note: If your query is on an indexed field, do not use tailable cursors, but instead, use a regular cursor. Keep track of the last value of the indexed field returned by the query. To retrieve the newly added documents, query the collection again using the last value of the indexed field in the query criteria, as in the following example:

```
db.<collection>.find( { indexedField: { $gt: <lastvalue> } } )
```

Consider the following behaviors related to tailable cursors:

- Tailable cursors do not use indexes and return documents in *natural order*.
- Because tailable cursors do not use indexes, the initial scan for the query may be expensive; but, after initially exhausting the cursor, subsequent retrievals of the newly added documents are inexpensive.
- Tailable cursors may become *dead*, or invalid, if either:
  - the query returns no match.
  - the cursor returns the document at the "end" of the collection and then the application deletes those document.

  A *dead* cursor has an id of `0`.

See your *driver documentation* (page 575) for the driver-specific method to specify the tailable cursor. For more information on the details of specifying a tailable cursor, see MongoDB wire protocol documentation.

36.2.2 C++ Example

The `tail` function uses a tailable cursor to output the results from a query to a capped collection:

- The function handles the case of the dead cursor by having the query be inside a loop.
- To periodically check for new data, the `cursor->more()` statement is also inside a loop.

```
#include "client/dbclient.h"

using namespace mongo;

/*
 * Example of a tailable cursor.
 * The function "tails" the capped collection (ns) and output elements as they are added.
 * The function also handles the possibility of a dead cursor by tracking the field 'insertDate'.
 * New documents are added with increasing values of 'insertDate'.
 */

void tail(DBClientBase& conn, const char *ns) {

    BSONElement lastValue = minKey.firstElement();

    Query query = Query().hint( BSON( "$natural" << 1 ) );

    while ( 1 ) {
        auto_ptr<DBClientCursor> c =
            conn.query(ns, query, 0, 0, 0,
                       QueryOption_CursorTailable | QueryOption_AwaitData );

        while ( 1 ) {
            if ( !c->more() ) {

                if ( c->isDead() ) {
                    break;
                }
```

```
                continue;
            }

            BSONObj o = c->next();
            lastValue = o["insertDate"];
            cout << o.toString() << endl;
        }

        query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );
    }
}
```

The `tail` function performs the following actions:

- Initialize the `lastValue` variable, which tracks the last accessed value. The function will use the `lastValue` if the cursor becomes *invalid* and `tail` needs to restart the query. Use `hint()` (page 985) to ensure that the query uses the `$natural` (page 833) order.
- In an outer `while(1)` loop,
  - Query the capped collection and return a tailable cursor that blocks for several seconds waiting for new documents

    ```
    auto_ptr<DBClientCursor> c =
         conn.query(ns, query, 0, 0, 0,
                    QueryOption_CursorTailable | QueryOption_AwaitData );
    ```

    * Specify the capped collection using `ns` as an argument to the function.
    * Set the `QueryOption_CursorTailable` option to create a tailable cursor.
    * Set the `QueryOption_AwaitData` option so that the returned cursor blocks for a few seconds to wait for data.
  - In an inner `while (1)` loop, read the documents from the cursor:
    * If the cursor has no more documents and is not invalid, loop the inner `while` loop to recheck for more documents.
    * If the cursor has no more documents and is dead, break the inner `while` loop.
    * If the cursor has documents:
      · output the document,
      · update the `lastValue` value,
      · and loop the inner `while (1)` loop to recheck for more documents.
  - If the logic breaks out of the inner `while (1)` loop and the cursor is invalid:
    * Use the `lastValue` value to create a new query condition that matches documents added after the `lastValue`. Explicitly ensure `$natural` order with the `hint()` method:

      ```
      query = QUERY( "insertDate" << GT << lastValue ).hint( BSON( "$natural" << 1 ) );
      ```

    * Loop through the outer `while (1)` loop to re-query with the new query condition and repeat.

See also:

Detailed blog post on tailable cursor

36.3 Isolate Sequence of Operations

36.3.1 Overview

Write operations are atomic on the level of a single document: no single write operation can atomically affect more than one document or more than one collection.

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

No other operations are atomic; however, you can *isolate* a single write operation that affects multiple documents using the *isolation operator* (page 822).

This document describes one method of updating documents *only* if the local copy of the document reflects the current state of the document in the database. In addition the following methods provide a way to manage isolated sequences of operations:

- the `findAndModify()` (page 952) provides an isolated query and modify operation.
- *Perform Two Phase Commits* (page 585)
- Create a *unique index* (page 314), to ensure that a key doesn't exist when you insert it.

36.3.2 Update if Current

In this pattern, you will:

- query for a document,
- modify the fields in that document
- and update the fields of a document *only if* the fields have not changed in the collection since the query.

Consider the following example in JavaScript which attempts to update the `qty` field of a document in the `products` collection:

```
var myCollection = db.products;
var myDocument = myCollection.findOne( { sku: 'abc123' } );

if (myDocument) {

   var oldQty = myDocument.qty;

   if (myDocument.qty < 10) {
       myDocument.qty *= 4;
   } else if ( myDocument.qty < 20 ) {
       myDocument.qty *= 3;
   } else {
       myDocument.qty *= 2;
   }

   myCollection.update(
      {
        _id: myDocument._id,
        qty: oldQty
      },
      {
        $set: { qty: myDocument.qty }
```

```
23          }
24      )
25
26      var err = db.getLastErrorObj();
27
28      if ( err && err.code ) {
29          print("unexpected error updating document: " + tojson( err ));
30      } else if ( err.n == 0 ) {
31          print("No update: no matching document for { _id: " + myDocument._id + ", qty: " + oldQty + "
32      }
33
34  }
```

Your application may require some modifications of this pattern, such as:

- Use the entire document as the query in lines 18 and 19, to generalize the operation and guarantee that the original document was not modified, rather than ensuring that as single field was not changed.
- Add a version variable to the document that applications increment upon each update operation to the documents. Use this version variable in the query expression. You must be able to ensure that *all* clients that connect to your database obey this constraint.
- Use `$set` (page 814) in the update expression to modify only your fields and prevent overriding other fields.
- Use one of the methods described in *Create an Auto-Incrementing Sequence Field* (page 594).

36.4 Create an Auto-Incrementing Sequence Field

36.4.1 Synopsis

MongoDB reserves the `_id` field in the top level of all documents as a primary key. `_id` must be unique, and always has an index with a *unique constraint* (page 314). However, except for the unique constraint you can use any value for the `_id` field in your collections. This tutorial describes two methods for creating an incrementing sequence number for the `_id` field using the following:

- *A Counters Collection* (page 594)
- *Optimistic Loop* (page 596)

> **Warning:** Generally in MongoDB, you would not use an auto-increment pattern for the `_id` field, or any field, because it does not scale for databases with large numbers of documents. Typically the default value *ObjectId* is more ideal for the `_id`.

A Counters Collection

Use a separate `counters` collection to track the *last* number sequence used. The `_id` field contains the sequence name and the `seq` field contains the last value of the sequence.

1. Insert into the `counters` collection, the initial value for the `userid`:

```
db.counters.insert(
   {
      _id: "userid",
      seq: 0
   }
)
```

2. Create a `getNextSequence` function that accepts a `name` of the sequence. The function uses the `findAndModify()` (page 952) method to atomically increment the `seq` value and return this new value:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true
          }
   );

   return ret.seq;
}
```

3. Use this `getNextSequence()` function during `insert()` (page 961).

```
db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Sarah C."
   }
)

db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Bob D."
   }
)
```

You can verify the results with `find()` (page 951):

```
db.users.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id : 1,
  name : "Sarah C."
}
{
  _id : 2,
  name : "Bob D."
}
```

Note: When `findAndModify()` (page 952) includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients each invoke the method with the same query condition and these methods complete the find phase before any of methods perform the modify phase, these methods could insert the same document.

In the `counters` collection example, the query field is the `_id` field, which always has a unique index. Consider that the `findAndModify()` (page 952) includes the `upsert: true` option, as in the following modified example:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true,
```

```
            upsert: true
        }
    );

    return ret.seq;
}
```

If multiple clients were to invoke the `getNextSequence()` method with the same `name` parameter, then the methods would observe one of the following behaviors:

- Exactly one `findAndModify()` (page 952) would successfully insert a new document.
- Zero or more `findAndModify()` (page 952) methods would update the newly inserted document.
- Zero or more `findAndModify()` (page 952) methods would fail when they attempted to insert a duplicate.

If the method fails due to a unique index constraint violation, retry the method. Absent a delete of the document, the retry should not fail.

Optimistic Loop

In this pattern, an *Optimistic Loop* calculates the incremented `_id` value and attempts to insert a document with the calculated `_id` value. If the insert is successful, the loop ends. Otherwise, the loop will iterate through possible `_id` values until the insert is successful.

1. Create a function named `insertDocument` that performs the "insert if not present" loop. The function wraps the `insert()` (page 961) method and takes a `doc` and a `targetCollection` arguments.

```
function insertDocument(doc, targetCollection) {

    while (1) {

        var cursor = targetCollection.find( {}, { _id: 1 } ).sort( { _id: -1 } ).limit(1);

        var seq = cursor.hasNext() ? cursor.next()._id + 1 : 1;

        doc._id = seq;

        targetCollection.insert(doc);

        var err = db.getLastErrorObj();

        if( err && err.code ) {
            if( err.code == 11000 /* dup key */ )
                continue;
            else
                print( "unexpected error inserting data: " + tojson( err ) );
        }

        break;
    }
}
```

The `while (1)` loop performs the following actions:

- Queries the `targetCollection` for the document with the maximum `_id` value.
- Determines the next sequence value for `_id` by:

    – adding `1` to the returned `_id` value if the returned cursor points to a document.
    – otherwise: it sets the next sequence value to `1` if the returned cursor points to no document.

  • For the `doc` to insert, set its `_id` field to the calculated sequence value `seq`.
  • Insert the `doc` into the `targetCollection`.
  • If the insert operation errors with duplicate key, repeat the loop. Otherwise, if the insert operation encounters some other error or if the operation succeeds, break out of the loop.

2. Use the `insertDocument()` function to perform an insert:

```
var myCollection = db.users2;

insertDocument(
   {
     name: "Grace H."
   },
   myCollection
);

insertDocument(
   {
     name: "Ted R."
   },
   myCollection
)
```

   You can verify the results with `find()` (page 951):

```
db.users2.find()
```

   The `_id` fields contain incrementing sequence values:

```
{
  _id: 1,
  name: "Grace H."
}
{
  _id : 2,
  "name" : "Ted R."
}
```

The `while` loop may iterate many times in collections with larger insert volumes.

36.5 Limit Number of Elements in an Array after an Update

New in version 2.4.

36.5.1 Synopsis

Consider an application where users may submit many scores (e.g. for a test), but the application only needs to track the top three test scores.

This pattern uses the `$push` (page 818) operator with the `$each` (page 819), `$sort` (page 820), and `$slice` (page 819) modifiers to sort and maintain an array of fixed size.

Important: The array elements must be documents in order to use the `$sort` (page 820) modifier.

36.5.2 Pattern

Consider the following document in the collection `students`:

```
{
  _id: 1,
  scores: [
            { attempt: 1, score: 10 },
            { attempt: 2 , score:8 }
          ]
}
```

The following update uses the `$push` (page 818) operator with:

- the `$each` (page 819) modifier to append to the array 2 new elements,
- the `$sort` (page 820) modifier to order the elements by ascending (`1`) score, and
- the `$slice` (page 819) modifier to keep the last `3` elements of the ordered array.

```
db.students.update(
                    { _id: 1 },
                    { $push: { scores: { $each : [
                                                   { attempt: 3, score: 7 },
                                                   { attempt: 4, score: 4 }
                                                 ],
                                         $sort: { score: 1 },
                                         $slice: -3
                                       }
                             }
                    }
                  )
```

Note: When using the `$sort` (page 820) modifier on the array element, access the field in the subdocument element directly instead of using the *dot notation* on the array field.

After the operation, the document contains the only the top 3 scores in the `scores` array:

```
{
   "_id" : 1,
   "scores" : [
                { "attempt" : 3, "score" : 7 },
                { "attempt" : 2, "score" : 8 },
                { "attempt" : 1, "score" : 10 }
             ]
}
```

See also:

- `$push` (page 818) operator,
- `$each` (page 819) modifier,
- `$sort` (page 820) modifier, and
- `$slice` (page 819) modifier.

36.6 Expire Data from Collections by Setting TTL

New in version 2.2.

This document provides an introduction 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` (page 1049) 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.

36.6.1 Background

Collections expire by way of a special index of a field with date values in conjunction with a background thread in `mongod` (page 1049) 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` (page 949) method in conjunction with a TTL value in seconds to create an expiring collection. Collections with an index that expires data set the `usePowerOf2Sizes` (page 892) collection flag. As a result, 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.

Important: All collections with an index using the `expireAfterSeconds` option have `usePowerOf2Sizes` (page 892) enabled. Users cannot modify this setting.

Note: When the TTL thread is active, you will see *delete* (page 101) operations in the output of `db.currentOp()` (page 998) or in the data collected by the *database profiler* (page 175).

36.6.2 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.
- documents that omit the indexed field will never 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 *every 60 seconds*. 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` (page 1049) instance.

36.6.3 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` (page 1066) 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.

36.6.4 Replication

The TTL background thread *only* runs on *primary* members of *replica sets*. *Secondaries* members will replicate deletion operations from the primaries.

Part XI

The `mongo` Shell

The `mongo` (page 1066) shell is an interactive JavaScript shell for MongoDB, and is part of all MongoDB distributions. This section provides an introduction to the shell, and outlines key functions, operations, and use of the `mongo` (page 1066) shell.

Most examples in the *MongoDB Manual* (page 1) use the `mongo` (page 1066) shell; however, many *drivers* (page 575) provide similar interfaces to MongoDB.

Getting Started with the mongo Shell

This document provides a basic introduction to using the mongo (page 1066) shell. See *Install MongoDB* (page 3) for instructions on installing MongoDB for your system.

37.1 Start the mongo Shell

To start the mongo (page 1066) shell and connect to your *MongoDB* (page 1049) instance running on **localhost** with **default port**:

1. Go to your `<mongodb installation dir>`:

   ```
   cd <mongodb installation dir>
   ```

2. Type `./bin/mongo` to start mongo (page 1066):

   ```
   ./bin/mongo
   ```

   If you have added the `<mongodb installation dir>/bin` to the `PATH` environment variable, you can just type `mongo` instead of `./bin/mongo`.

3. To display the database you are using, type `db`:

   ```
   db
   ```

   The command should return `test`, which is the default database. To switch databases, issue the `use <db>` command, as in the following example:

   ```
   use <database>
   ```

   To list the available databases, use the command `show dbs`. See also *How can I access different databases temporarily?* (page 745) to access a different database from the current database without switching your current database context (i.e. `db..`)

To start the mongo (page 1066) shell with other options, see *examples of starting up mongo* (page 1071) and *mongo reference* (page 1066) which provides details on the available options.

Note: When starting, mongo (page 1066) checks the user's HOME (page 1070) directory for a JavaScript file named *.mongorc.js* (page 1069). If found, mongo (page 1066) interprets the content of `.mongorc.js` before displaying the prompt for the first time. If you use the shell to evaluate a JavaScript file or expression, either by using the `--eval` option on the command line or by specifying *a .js file to mongo* (page 1069), mongo (page 1066) will read the `.mongorc.js` file *after* the JavaScript has finished processing.

37.2 Executing Queries

From the `mongo` (page 1066) shell, you can use the *shell methods* (page 944) to run queries, as in the following example:

```
db.<collection>.find()
```

- The `db` refers to the current database.
- The `<collection>` is the name of the collection to query. See *Collection Help* (page 614) to list the available collections.

  If the `mongo` (page 1066) shell does not accept the name of the collection, for instance if the name contains a space, hyphen, or starts with a number, you can use an alternate syntax to refer to the collection, as in the following:

  ```
  db["3test"].find()

  db.getCollection("3test").find()
  ```

- The `find()` (page 951) method is the JavaScript method to retrieve documents from `<collection>`. The `find()` (page 951) method returns a *cursor* to the results; however, in the `mongo` (page 1066) shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents that match the query. The `mongo` (page 1066) shell will prompt `Type it` to iterate another 20 times.

  You can set the `DBQuery.shellBatchSize` attribute to change the number of iteration from the default value `20`, as in the following example which sets it to `10`:

  ```
  DBQuery.shellBatchSize = 10;
  ```

  For more information and examples on cursor handling in the `mongo` (page 1066) shell, see *Cursors* (page 49).

  See also *Cursor Help* (page 614) for list of cursor help in the `mongo` (page 1066) shell.

For more documentation of basic MongoDB operations in the `mongo` (page 1066) shell, see:

- *Getting Started with MongoDB* (page 29)
- *mongo Shell Quick Reference* (page 619)
- *Create* (page 75)
- *Read* (page 83)
- *Update* (page 93)
- *Delete* (page 101)
- *Indexing Operations* (page 329)
- *Read Operations* (page 41)
- *Write Operations* (page 53)

37.3 Print

The `mongo` (page 1066) shell automatically prints the results of the `find()` (page 951) method if the returned cursor is not assigned to a variable. To format the result, you can add the `.pretty()` to the operation, as in the following:

```
db.<collection>.find().pretty()
```

In addition, you can use the following explicit print methods in the `mongo` (page 1066) shell:

- `print()` to print without formatting
- `print(tojson(<obj>))` to print with *JSON* formatting and equivalent to `printjson()`
- `printjson()` to print with *JSON* formatting and equivalent to `print(tojson(<obj>))`

37.4 Use a Custom Prompt

You may modify the content of the prompt by creating the variable `prompt` in the shell. The prompt variable can hold strings as well as any arbitrary JavaScript. If `prompt` holds a function that returns a string, `mongo` (page 1066) can display dynamic information in each prompt. Consider the following examples:

Example

Create a prompt with the number of commands issued in the current session, define the following variables:

```
cmdCount = 1;
prompt = function() {
             return (cmdCount++) + "> ";
         }
```

The prompt would then resemble the following:

```
1> db.collection.find()
2> show collections
3>
```

Example

To create a `mongo` (page 1066) shell prompt in the form of `<database>@<hostname>$` define the following variables:

```
host = db.serverStatus().host;

prompt = function() {
             return db+"@"+host+"$ ";
         }
```

The prompt would then resemble the following:

```
<database>@<hostname>$ use records
switched to db records
records@<hostname>$
```

Example

To create a `mongo` (page 1066) shell prompt that contains the system up time *and* the number of documents in the current database, define the following prompt variable:

```
prompt = function() {
             return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > ";
         }
```

The prompt would then resemble the following:

```
Uptime:5897 Documents:6 > db.people.save({name : "James"});
Uptime:5948 Documents:7 >
```

37.5 Use an External Editor in the mongo Shell

New in version 2.2.

In the `mongo` (page 1066) shell you can use the `edit` operation to edit a function or variable in an external editor. The `edit` operation uses the value of your environments `EDITOR` variable.

At your system prompt you can define the `EDITOR` variable and start `mongo` (page 1066) with the following two operations:

```
export EDITOR=vim
mongo
```

Then, consider the following example shell session:

```
MongoDB shell version: 2.2.0
> function f() {}
> edit f
> f
function f() {
    print("this really works");
}
> f()
this really works
> o = {}
{ }
> edit o
> o
{ "soDoes" : "this" }
>
```

Note: As `mongo` (page 1066) shell interprets code edited in an external editor, it may modify code in functions, depending on the JavaScript compiler. For `mongo` (page 1066) may convert `1+1` to `2` or remove comments. The actual changes affect only the appearance of the code and will vary based on the version of JavaScript used but will not affect the semantics of the code.

37.6 Exit the Shell

To exit the shell, type `quit()` or use the `<Ctrl-c>` shortcut.

CHAPTER 38

Data Types in the mongo Shell

MongoDB *BSON* provide support for additional data types than *JSON*. *Drivers* (page 575) provide native support for these data types in host languages and the mongo (page 1066) shell also provides several helper classes to support the use of these data types in the mongo (page 1066) JavaScript shell. See *MongoDB Extended JSON* (page 1147) for additional information.

38.1 Date

The mongo (page 1066) shell provides various options to return the date, either as a string or as an object:

- `Date()` method which returns the current date as a string.
- `Date()` constructor which returns an `ISODate` object when used with the `new` operator.
- `ISODate()` constructor which returns an `ISODate` object when used with *or* without the `new` operator.

Consider the following examples:

- To return the date as a string, use the `Date()` method, as in the following example:

  ```
  var myDateString = Date();
  ```

  – To print the value of the variable, type the variable name in the shell, as in the following:

    ```
    myDateString
    ```

    The result is the value of `myDateString`:

    ```
    Wed Dec 19 2012 01:03:25 GMT-0500 (EST)
    ```

  – To verify the type, use the `typeof` operator, as in the following:

    ```
    typeof myDateString
    ```

    The operation returns `string`.

- To get the date as an `ISODate` object, instantiate a new instance using the `Date()` constructor with the `new` operator, as in the following example:

  ```
  var myDateObject = new Date();
  ```

  – To print the value of the variable, type the variable name in the shell, as in the following:

    ```
    myDateObject
    ```

The result is the value of `myDateObject`:

```
ISODate("2012-12-19T06:01:17.171Z")
```

– To verify the type, use the `typeof` operator, as in the following:

```
typeof myDateObject
```

The operation returns `object`.

- To get the date as an `ISODate` object, instantiate a new instance using the `ISODate()` constructor *without* the `new` operator, as in the following example:

```
var myDateObject2 = ISODate();
```

You can use the `new` operator with the `ISODate()` constructor as well.

– To print the value of the variable, type the variable name in the shell, as in the following:

```
myDateObject2
```

The result is the value of `myDateObject2`:

```
ISODate("2012-12-19T06:15:33.035Z")
```

– To verify the type, use the `typeof` operator, as in the following:

```
typeof myDateObject2
```

The operation returns `object`.

38.2 ObjectId

The `mongo` (page 1066) shell provides the `ObjectId()` wrapper class around *ObjectId* data types. To generate a new ObjectId, use the following operation in the `mongo` (page 1066) shell:

```
new ObjectId
```

See

ObjectId (page 68) for full documentation of ObjectIds in MongoDB.

38.3 NumberLong

By default, the `mongo` (page 1066) shell treats all numbers as floating-point values. The `mongo` (page 1066) shell provides the `NumberLong()` class to handle 64-bit integers.

The `NumberLong()` constructor accepts the long as a string:

```
NumberLong("2090845886852")
```

The following examples use the `NumberLong()` class to write to the collection:

```
db.collection.insert( { _id: 10, calc: NumberLong("2090845886852") } )
db.collection.update( { _id: 10 },
                      { $set:  { calc: NumberLong("2555555000000") } } )
```

```
db.collection.update( { _id: 10 },
                      { $inc: { calc: NumberLong(5) } } )
```

Retrieve the document to verify:

```
db.collection.findOne( { _id: 10 } )
```

In the returned document, the `calc` field contains a `NumberLong` object:

```
{ "_id" : 10, "calc" : NumberLong("2555555000005") }
```

If you use the `$inc` (page 810) to increment the value of a field that contains a `NumberLong` object by a **float**, the data type changes to a floating point value, as in the following example:

1. Use `$inc` (page 810) to increment the `calc` field by `5`, which the `mongo` (page 1066) shell treats as a float:

   ```
   db.collection.update( { _id: 10 },
                         { $inc: { calc: 5 } } )
   ```

2. Retrieve the updated document:

   ```
   db.collection.findOne( { _id: 10 } )
   ```

   In the updated document, the `calc` field contains a floating point value:

   ```
   { "_id" : 10, "calc" : 2555555000010 }
   ```

38.4 NumberInt

By default, the `mongo` (page 1066) shell treats all numbers as floating-point values. The `mongo` (page 1066) shell provides the `NumberInt()` constructor to explicitly specify 32-bit integers.

CHAPTER 39

Access the mongo Shell Help Information

In addition to the documentation in the *MongoDB Manual* (page 1), the mongo (page 1066) shell provides some additional information in its "online" help system. This document provides an overview of accessing this help information.

See also:

- *mongo Manual Page* (page 1066)
- *The mongo Shell* (page 603) (*MongoDB Manual* (page 1) section on the shell.)
- *mongo Shell Quick Reference* (page 619).

39.1 Command Line Help

To see the list of options and help for starting the mongo (page 1066) shell, use the *--help* option from the command line:

```
mongo --help
```

39.2 Shell Help

To see the list of help, in the mongo (page 1066) shell, type `help`:

```
help
```

39.3 Database Help

- To see the list of databases on the server, use the `show dbs` command:

  ```
  show dbs
  ```

  New in version 2.4: `show databases` is now an alias for `show dbs`

- To see the list of help for methods you can use on the `db` object, call the `db.help()` (page 1008) method:

  ```
  db.help()
  ```

- To see the implementation of a method in the shell, type the `db.<method name>` without the parenthesis (`()`), as in the following example which will return the implementation of the method `db.addUser()` (page 993):

  ```
  db.addUser
  ```

39.4 Collection Help

- To see the list of collections in the current database, use the `show collections` command:

  ```
  show collections
  ```

- To see the help for methods available on the collection objects (e.g. `db.<collection>`), use the `db.<collection>.help()` method:

  ```
  db.collection.help()
  ```

  `<collection>` can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the collection method implementation, type the `db.<collection>.<method>` name without the parenthesis (`()`), as in the following example which will return the implementation of the `save()` (page 972) method:

  ```
  db.collection.save
  ```

39.5 Cursor Help

When you perform *read operations* (page 41) with the `find()` (page 951) method in the `mongo` (page 1066) shell, you can use various cursor methods to modify the `find()` (page 951) behavior and various JavaScript methods to handle the cursor returned from the `find()` (page 951) method.

- To list the available modifier and cursor handling methods, use the `db.collection.find().help()` command:

  ```
  db.collection.find().help()
  ```

  `<collection>` can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the implementation of the cursor method, type the `db.<collection>.find().<method>` name without the parenthesis (`()`), as in the following example which will return the implementation of the `toArray()` method:

  ```
  db.collection.find().toArray
  ```

Some useful methods for handling cursors are:

- `hasNext()` (page 985) which checks whether the cursor has more documents to return.
- `next()` (page 989) which returns the next document and advances the cursor position forward by one.
- `forEach(<function>)` (page 984) which iterates the whole cursor and applies the `<function>` to each document returned by the cursor. The `<function>` expects a single argument which corresponds to the document from each iteration.

For examples on iterating a cursor and retrieving the documents from the cursor, see *cursor handling* (page 49). See also *Cursor* (page 977) for all available cursor methods.

39.6 Type Help

To get a list of the wrapper classes available in the `mongo` (page 1066) shell, such as `BinData()`, type `help misc` in the `mongo` (page 1066) shell:

```
help misc
```

Write Scripts for the mongo Shell

You can write scripts for the `mongo` (page 1066) shell in JavaScript that manipulate data in MongoDB or perform administrative operation. For more information about the `mongo` (page 1066) shell see *The mongo Shell* (page 603), and see the *Running .js files via a mongo shell Instance on the Server* (page 581) section for more information about using these `mongo` (page 1066) script.

This tutorial provides an introduction to writing JavaScript that uses the `mongo` (page 1066) shell to access MongoDB.

40.1 Opening New Connections

From the `mongo` (page 1066) shell or from a JavaScript file, you can instantiate database connections using the `Mongo()` (page 1036) constructor:

```
new Mongo()
new Mongo(<host>)
new Mongo(<host:port>)
```

Consider the following example that instantiates a new connection to the MongoDB instance running on localhost on the default port and sets the global `db` variable to `myDatabase` using the `getDB()` (page 1033) method:

```
conn = new Mongo();
db = conn.getDB("myDatabase");
```

Additionally, you can use the `connect()` method to connect to the MongoDB instance. The following example connects to the MongoDB instance that is running on `localhost` with the non-default port `27020` and set the global `db` variable:

```
db = connect("localhost:27020/myDatabase");
```

If you create new connections inside a *JavaScript file* (page 618):

- To set the `db` global variable, use the `getDB()` (page 1033) method or the `connect()` method. You can assign the database reference to a variable other than `db`.
- Additionally, inside the script, you would need to call `db.getLastErrorObj()` (page 1005) or `db.getLastError()` (page 1005) explicitly to wait for the result of *write operations* (page 53).
- You **cannot** use any shell helper (e.g. `use <dbname>`, `show dbs`, etc.) inside the JavaScript file because they are not valid JavaScript.

  The following table maps the most common `mongo` (page 1066) shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.system.users.find()`
`show log <logname>`	`db.adminCommand( { 'getLog' : '<logname>' } )`
`show logs`	`db.adminCommand( { 'getLog' : '*' } )`

40.2 Scripting

From the command line, use `mongo` (page 1066) to evaluate JavaScript.

40.2.1 `--eval` option

Use the `--eval` option to `mongo` (page 1066) to pass the shell a JavaScript fragment, as in the following:

```
mongo test --eval "printjson(db.getCollectionNames())"
```

This returns the output of `db.getCollectionNames()` (page 1005) using the `mongo` (page 1066) shell connected to the `mongod` (page 1049) or `mongos` (page 1061) instance running on port `27017` on the `localhost` interface.

40.2.2 Evaluate a JavaScript file

You can specify a `.js` file to the `mongo` (page 1066) shell, and `mongo` (page 1066) will evaluate the javascript directly. Consider the following example:

```
mongo localhost:27017/test myjsfile.js
```

This operation evaluates the `myjsfile.js` script in a `mongo` (page 1066) shell that connects to the `test` *database* on the `mongod` (page 1049) instance accessible via the `localhost` interface on port `27017`.

Alternately, you can specify the mongodb connection parameters inside of the javascript file using the `Mongo()` constructor. See *Opening New Connections* (page 617) for more information.

CHAPTER 41

mongo Shell Quick Reference

41.1 mongo Shell Command History

You can retrieve previous commands issued in the mongo (page 1066) shell with the up and down arrow keys. Command history is stored in ~/.dbshell file. See *.dbshell* (page 1069) for more information.

41.2 Command Line Options

The mongo (page 1066) executable can be started with numerous options. See *mongo executable* (page 1066) page for details on all available options.

The following table displays some common options for mongo (page 1066):

Option	Description
`--help`	Show command line options
`--nodb`	Start mongo (page 1066) shell without connecting to a database. To connect later, see *Opening New Connections* (page 617).
`--shell`	Used in conjunction with a JavaScript file (i.e. *<file.js>* (page 1069)) to continue in the mongo (page 1066) shell after running the JavaScript file. See *JavaScript file* (page 618) for an example.

41.3 Command Helpers

The mongo (page 1066) shell provides various help. The following table displays some common help methods and commands:

Help Methods and Commands	Description
`help`	Show help.
`db.help()`	Show help for database methods.
`db.<collection>.help()`	Show help on collection methods. The `<collection>` can be the name of an existing collection or a non-existing collection.
`show dbs`	Print a list of all databases on the server.
`use <db>`	Switch current database to `<db>`. The `mongo` (page 1066) shell variable `db` is set to the current database.
`show collections`	Print a list of all collections for current database
`show users`	Print a list of users for current database.
`show profile`	Print the five most recent operations that took 1 millisecond or more. See documentation on the *database profiler* (page 175) for more information.
`show databases`	New in version 2.4: Print a list of all available databases.

41.4 Basic Shell JavaScript Operations

The `mongo` (page 1066) shell provides numerous *mongo Shell Methods* (page 944) methods for database operations.

In the `mongo` (page 1066) shell, `db` is the variable that references the current database. The variable is automatically set to the default database `test` or is set when you use the `use <db>` to switch current database.

The following table displays some common JavaScript operations:

JavaScript Database Operations	Description
db.auth() (page 995)	If running in secure mode, authenticate the user.

coll = db.<collection>	Set a specific collection in the current database to a variable coll, as in the following example:
	coll = db.myCollection;
	You can perform operations on the myCollection using the variable, as in the following example:
	coll.find();

db.collection.find() (page 951)	Find all documents in the collection and returns a cursor. See the *Read* (page 83) and *Read Operations* (page 41) for more information and examples. See *Cursors* (page 49) for additional information on cursor handling in the mongo (page 1066) shell.
db.collection.insert() (page 961)	Insert a new document into the collection.
db.collection.update() (page 974)	Update an existing document in the collection. See *Update* (page 93) for more information.
db.collection.save() (page 972)	Insert either a new document or update an existing document in the collection. See *Update* (page 93) for more information.
db.collection.remove() (page 970)	Delete documents from the collection. See *Delete* (page 101) for more information.
db.collection.drop() (page 948)	Drops or removes completely the collection.
db.collection.ensureIndex() (page 949)	Create a new index on the collection if the index does not exist; otherwise, the operation has no effect.
db.getSiblingDB() (page 1007) or db.getSisterDB()	Return a reference to another database using this same connection without explicitly switching the current database. This allows for cross database queries. See *How can I access different databases temporarily?* (page 745) for more information.

For more information on performing operations in the shell, see:

- *Create* (page 75)
- *Read* (page 83)
- *Update* (page 93)
- *Delete* (page 101)
- *Indexing Operations* (page 329)
- *Read Operations* (page 41)
- *Write Operations* (page 53)
- *mongo Shell Methods* (page 944)

41.5 Keyboard Shortcuts

Changed in version 2.2.

The mongo (page 1066) shell provides most keyboard shortcuts similar to those found in the bash shell or in Emacs. For some functions mongo (page 1066) provides multiple key bindings, to accommodate several familiar paradigms.

The following table enumerates the keystrokes supported by the mongo (page 1066) shell:

Keystroke	Function
Up-arrow	previous-history
Down-arrow	next-history
Home	beginning-of-line
End	end-of-line
Tab	autocomplete
Left-arrow	backward-character
Right-arrow	forward-character
Ctrl-left-arrow	backward-word
Ctrl-right-arrow	forward-word
Meta-left-arrow	backward-word
Meta-right-arrow	forward-word
Ctrl-A	beginning-of-line
Ctrl-B	backward-char
Ctrl-C	exit-shell
Ctrl-D	delete-char (or exit shell)
Ctrl-E	end-of-line
Ctrl-F	forward-char
Ctrl-G	abort
Ctrl-J	accept-line
Ctrl-K	kill-line
Ctrl-L	clear-screen
Ctrl-M	accept-line
Ctrl-N	next-history
Ctrl-P	previous-history
Ctrl-R	reverse-search-history
Ctrl-S	forward-search-history
Ctrl-T	transpose-chars
Ctrl-U	unix-line-discard
Ctrl-W	unix-word-rubout
Ctrl-Y	yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H (i.e. Backspace)	backward-delete-char
Ctrl-I (i.e. Tab)	complete
Meta-B	backward-word
Meta-C	capitalize-word
Meta-D	kill-word
Meta-F	forward-word
Meta-L	downcase-word
Meta-U	upcase-word
Meta-Y	yank-pop
Meta-[Backspace]	backward-kill-word
Meta-<	beginning-of-history
Meta->	end-of-history

41.6 Queries

In the `mongo` (page 1066) shell, perform read operations using the `db.collection.find()` (page 951) and `db.collection.findOne()` (page 955) methods.

The `db.collection.find()` (page 951) method returns a cursor object which the `mongo` (page 1066) shell iterates to print documents on screen. By default, `mongo` (page 1066) prints the first 20. The `mongo` (page 1066) shell will prompt the user to "`Type it`" to continue iterating the next 20 results.

The following table provides some common read operations in the `mongo` (page 1066) shell:

Read Operations	Description

`db.collection.find(<query>)` (page 951)	Find the documents matching the `<query>` criteria in the collection. If the `<query>` criteria is not specified or is empty (i.e `{}`), the read operation selects all documents in the collection.
	The following example selects the documents in the `users` collection with the `name` field equal to `"Joe"`:
	`coll = db.users;`
	`coll.find( { name: "Joe" } );`
	For more information on specifying the `<query>` criteria, see *Query Document* (page 42).

`db.collection.find( <query>, <projection> )` (page 951)	Find documents matching the `<query>` criteria and return just specific fields in the `<projection>`.
	The following example selects all documents from the collection but returns only the `name` field and the `_id` field. The `_id` is always returned unless explicitly specified to not return.
	`coll = db.users;`
	`coll.find( { },`
	`{ name: true }`
	`);`
	For more information on specifying the `<projection>`, see *Result Projections* (page 45).

`db.collection.find().sort( <sort order> )` (page 991)	Return results in the specified `<sort order>`.
	The following example selects all documents from the collection and returns the results sorted by the `name` field in ascending order (`1`). Use `-1` for descending order:
	`coll = db.users;`
	`coll.find().sort( { name: 1 } );`

`db.collection.find( <query> ).sort( <sort order> )` (page 991)	Return the documents matching the `<query>` criteria in the specified `<sort order>`.
`db.collection.find( ... ).limit( <n> )` (page 985)	Limit result to `<n>` rows. Highly recommended if you need only a certain number of rows for best performance.
`db.collection.find( ... ).skip( <n> )` (page 990)	Skip `<n>` results.
`db.collection.count()` (page 946)	Returns total number of documents in the collection.
`db.collection.find( <query> ).count()` (page 978)	Returns the total number of documents that match the query. The `count()` (page 978) ignores `limit()` (page 985) and `skip()` (page 990). For example, if 100 records match but the limit is 10, `count()` (page 978) will return 100. This will be faster than iterating yourself, but still take time.

`db.collection.findOne( <query> )` (page 955)	Find and return a single document. Returns null if not found.
	The following example selects a single document in the `users` collection with the `name` field matches to `"Joe"`:
	`coll = db.users;`
	`coll.findOne( { name: "Joe" } );`
	Internally, the `findOne()` (page 955) method is the `find()` (page 951) method with a `limit(1)` (page 985).

See *Read* (page 83) and *Read Operations* (page 41) documentation for more information and examples. See *Query, Update and Projection Operators* (page 785) to specify other query operators.

41.7 Error Checking Methods

The `mongo` (page 1066) shell provides numerous *administrative database methods* (page 944), including error checking methods. These methods are:

Error Checking Methods	Description
`db.getLastError()` (page 1005)	Returns error message from the last operation.
`db.getLastErrorObj()` (page 1005)	Returns the error document from the last operation.

41.8 Administrative Command Helpers

The following table lists some common methods to support database administration:

JavaScript Database Administration Methods	Description
`db.cloneDatabase(<host>)` (page 996)	Clone the current database from the `<host>` specified. The `<host>` database instance must be in noauth mode.
`db.copyDatabase(<from>, <to>, <host>)` (page 996)	Copy the `<from>` database from the `<host>` to the `<to>` database on the current server. The `<host>` database instance must be in `noauth` mode.
`db.fromColl.renameCollection(<toColl>)` (page 971)	Rename collection from `fromColl` to `<toColl>`.
`db.repairDatabase()` (page 1011)	Repair and compact the current database. This operation can be very slow on large databases.
`db.addUser( <user>, <pwd> )` (page 993)	Add user to current database.
`db.getCollectionNames()` (page 1005)	Get the list of all collections in the current database.
`db.dropDatabase()` (page 1002)	Drops the current database.

See also *administrative database methods* (page 944) for a full list of methods.

41.9 Opening Additional Connections

You can create new connections within the `mongo` (page 1066) shell.

The following table displays the methods to create the connections:

JavaScript Connection Create Methods	Description
`db = connect("<host><:port>/<dbname>")`	Open a new database connection.
`conn = new Mongo()` `db = conn.getDB("dbname")`	Open a connection to a new server using `new Mongo()`. Use `getDB()` method of the connection to select a database.

See also *Opening New Connections* (page 617) for more information on the opening new connections from the `mongo` (page 1066) shell.

41.10 Miscellaneous

The following table displays some miscellaneous methods:

Method	Description
`Object.bsonsize(<document>)`	Prints the *BSON* size of an <document>

See the MongoDB JavaScript API Documentation for a full list of JavaScript methods .

41.11 Additional Resources

Consider the following reference material that addresses the `mongo` (page 1066) shell and its interface:

- *mongo* (page 1066)
- *mongo Shell Methods* (page 944)
- *Query, Update and Projection Operators* (page 785)
- *Database Commands* (page 833)
- *Aggregation Framework Reference* (page 263)

Additionally, the MongoDB source code repository includes a jstests directory which contains numerous `mongo` (page 1066) shell scripts.

The *Getting Started with MongoDB* (page 29) provides a general introduction to MongoDB using examples from the `mongo` (page 1066) shell. Additionally, the following documents from other sections address areas relevant to users of the `mongo` (page 1066) shell:

- *FAQ: The mongo Shell* (page 745)
- *mongo* (page 1066)
- *mongo Shell Methods* (page 944)

Furthermore, consider the following reference material that addresses the `mongo` (page 1066) shell and its interface:

- *Query, Update and Projection Operators* (page 785)
- *Database Commands* (page 833)
- *Aggregation Framework Reference* (page 263)
- *Query Modification Operators* (page 827)

Part XII

Use Cases

The use case documents introduce the patterns, designs, and operations used in application development with MongoDB. Each document provides 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 examples and general use cases.

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 697), provides a complete and fully developed application that you can build using MongoDB and popular Python web development tool kits.

CHAPTER 42

Operational Intelligence

As an introduction to the use of MongoDB for operational intelligence and real time analytics use, the document "*Storing Log Data* (page 631)" describes several ways and approaches to modeling and storing machine generated data with MongoDB. Then, "*Pre-Aggregated Reports* (page 641)" describes methods and strategies for processing data to generate aggregated reports from raw event-data. Finally "*Hierarchical Aggregation* (page 650)" 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.

42.1 Storing Log Data

42.1.1 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 641)

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.

42.1.2 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.MongoClient()
>>> 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, w=0)
```

By setting `w=0`, 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. See *Write Concern* (page 395) for more information.

If you want to ensure that MongoDB acknowledges inserts, you can pass `w=1` argument as follows:

```
>>> db.events.insert(event, w=1)
```

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 `w=1`.

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='majority')
```

This will force your application to acknowledge that the data has replicated to a majority of configured members of the *replica set*. You can combine options as well:

```
>>> db.events.insert(event, j=True, w='majority')
```

In this case, your application will wait for a successful journal commit on the *primary and* a replication acknowledgment from a majority of configured secondaries. 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 395) and `getLastError` (page 861).

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/manualapache_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 639) 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()` (page 949) 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()` (page 949) 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. `$gte` (page 787)) `2000-10-10` but before (i.e. `$lt` (page 788)) `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()` (page 949) 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()` (page 945) offers greater flexibility, capacity with less complexity than the existing `mapReduce` (page 840) and `group` (page 836) aggregation commands.

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': {
...                 'path': 1,
...                 'date': {
...                     'y': { '$year': '$time' },
```

[1] To translate statements from the *aggregation framework* (page 247) to SQL, you can consider the `$match` (page 266) equivalent to `WHERE`, `$project` (page 264) to `SELECT`, and `$group` (page 269) to `GROUP BY`.

```
...                        '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 `$match` (page 266) to limit the documents that the aggregation framework must process. `$match` (page 266) is similar to a `find()` (page 951) query.

   This operation selects all documents where the value of the `time` field represents a date that is on or after (i.e. `$gte` (page 787)) `2000-10-10` but before (i.e. `$lt` (page 788)) `2000-10-11`.

2. Uses the `$project` (page 264) 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 269) 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 `$project` (page 264) as the remaining fields.
   - the `hits` field use the `$sum` (page 276) 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 269) 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 639)." of this document for additional recommendations for using sharding.

See also:

"*Aggregation Framework* (page 247)"

Index Support

To optimize the aggregation operation, ensure that the initial `$match` (page 266) 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.

42.1.3 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:

Sharding (page 485) and *FAQ: Sharding with MongoDB* (page 753)

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.
- 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.

[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.
[5] Typically, it is difficult to use these kinds of shard keys in queries.

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 640) and *sharding by a random key* (page 639). 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 639) 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.

42.1.4 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:

[6] You must still calculate the value of this synthetic key in your application when you insert documents into your collection.

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 641) 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.

42.2 Pre-Aggregated Reports

42.2.1 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 631)."

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` (page 810) operations will allow you to calculate statistics, produce simple range-based queries, and generate filters to support time-series charts of aggregated data.

42.2.2 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 642)" 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 642)" 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`.

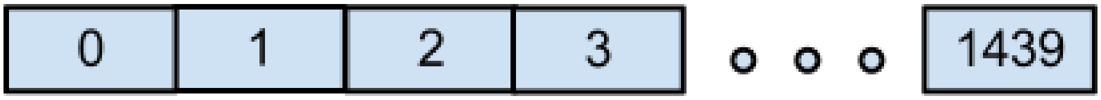


Figure 42.1: In order to update the value in minute #1349, MongoDB must skip over all 1349 entries before it.

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.

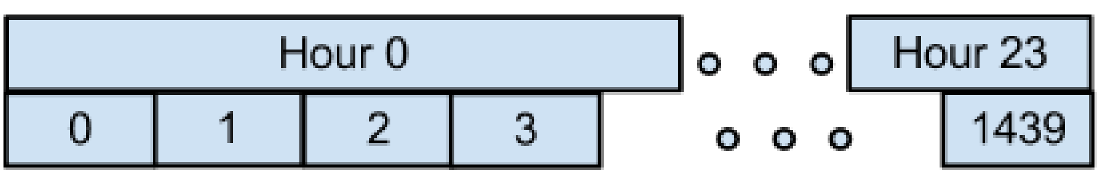


Figure 42.2: To update the value in minute #1349, MongoDB first skips the first 23 hours and then skips 59 minutes for only 82 skips as opposed to 1439 skips in the previous schema.

Separate Documents by Granularity Level

Pre-allocating documents (page 643) 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.

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,
        ... }
}
```

42.2.3 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 646) 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_metadata['date'].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/manualindex.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'}},
...     { 'hourly': 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.

42.2.4 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` (page 876) 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` (page 876) 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` (page 876) 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` (page 876) 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.

42.3 Hierarchical Aggregation

42.3.1 Overview

Background

If you collect a large amount of data, but do not *pre-aggregate* (page 641), 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 631)" 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` (page 840) *command*, in a hierarchical process. The following figure illustrates the input and output of each job:

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, `mapReduce` (page 840), and the *Map-Reduce* (page 291) page for more information on the Map-reduce data aggregation paradigm.

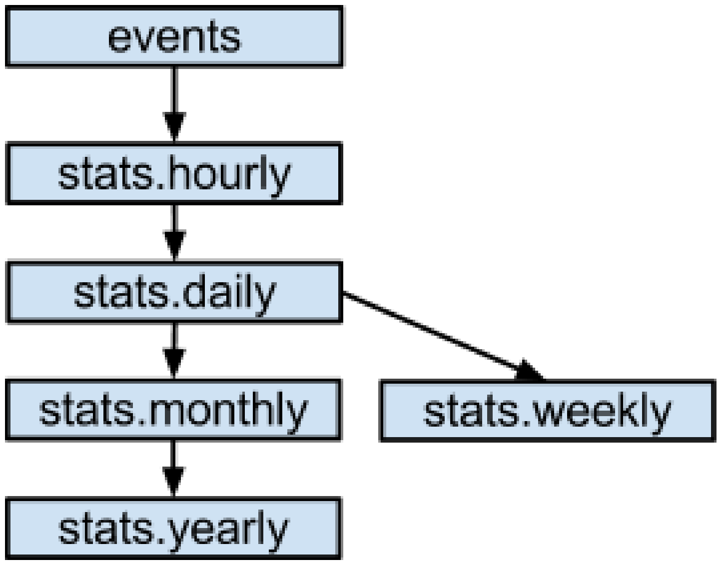


Figure 42.3: Hierarchy of data aggregation.

42.3.2 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.

42.3.3 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` (page 840) 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 `cutoff` 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(),
            0, 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.

42.3.4 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 43

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 659)" document describes methods and practices for modeling and managing a product catalog using MongoDB, while the "*Inventory Management* (page 667)" document introduces a pattern for handling interactions between inventory and users' shopping carts. Finally the "*Category Hierarchy* (page 673)" document describes methods for interacting with category hierarchies in MongoDB.

43.1 Product Catalog

43.1.1 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 660) and somewhat more flexible than *concrete table inheritance* (page 660). 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 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 660) 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.

43.1.2 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` (page 798) 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.

43.1.3 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 1061) 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 queries 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 unsplittable 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` (page 876) 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 1061) 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 1061) can route queries to specific shards. In these situations `mongos` (page 1061) 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 1061) 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.MongoClient(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.MongoClient(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 398)

43.2 Inventory Management

43.2.1 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:

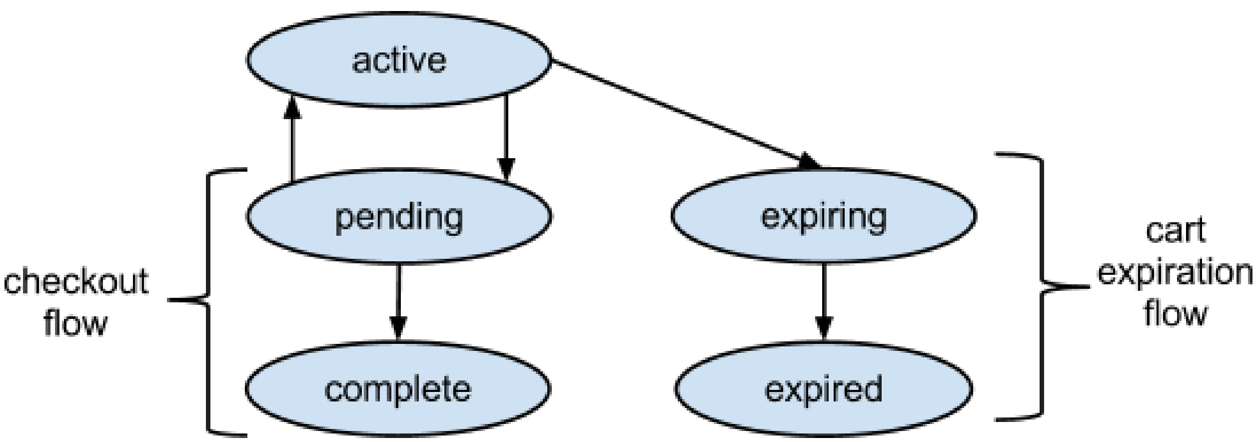
"*Product Catalog* (page 659)."

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 659)" 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.

43.2.2 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 } } },
        w=1)
    if not result['updatedExisting']:
        raise CartInactive()
```

```
    # Update the inventory
    result = db.inventory.update(
        {'_id':sku, 'qty': {'$gte': qty}},
        {'$inc': {'qty': -qty},
         '$push': {
             'carted': { 'qty': qty, 'cart_id':cart_id,
                         'timestamp': now } } },
        w=1)
    if not result['updatedExisting']:
        # Roll back our cart update
        db.cart.update(
            {'_id': cart_id },
            { '$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 the cart is "active" before adding an 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 `w=1` 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 },
        },
        w=1)
    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 } },
        w=1)
    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` (page 851) *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.

```python
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.

43.2.3 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 1061) to route all updates that select on `_id` to a single mongod (page 1049) 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 }
```

43.3 Category Hierarchy

43.3.1 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 659)"

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:

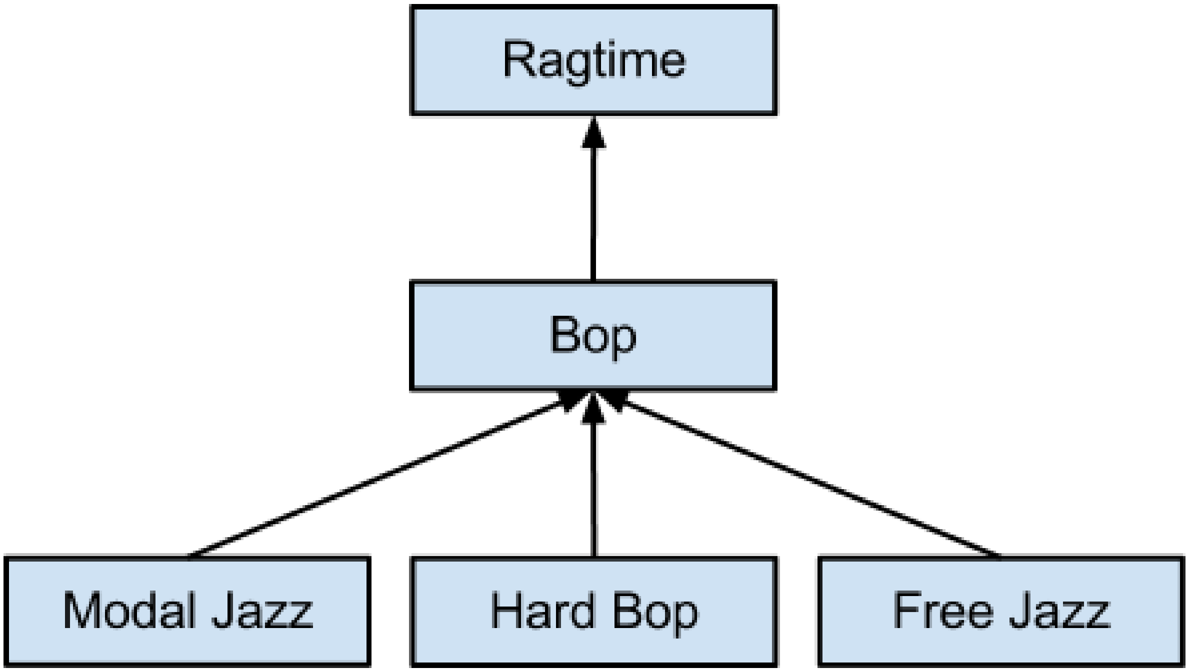


Figure 43.1: Initial category hierarchy

- 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.

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" } ]
}
```

43.3.2 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:

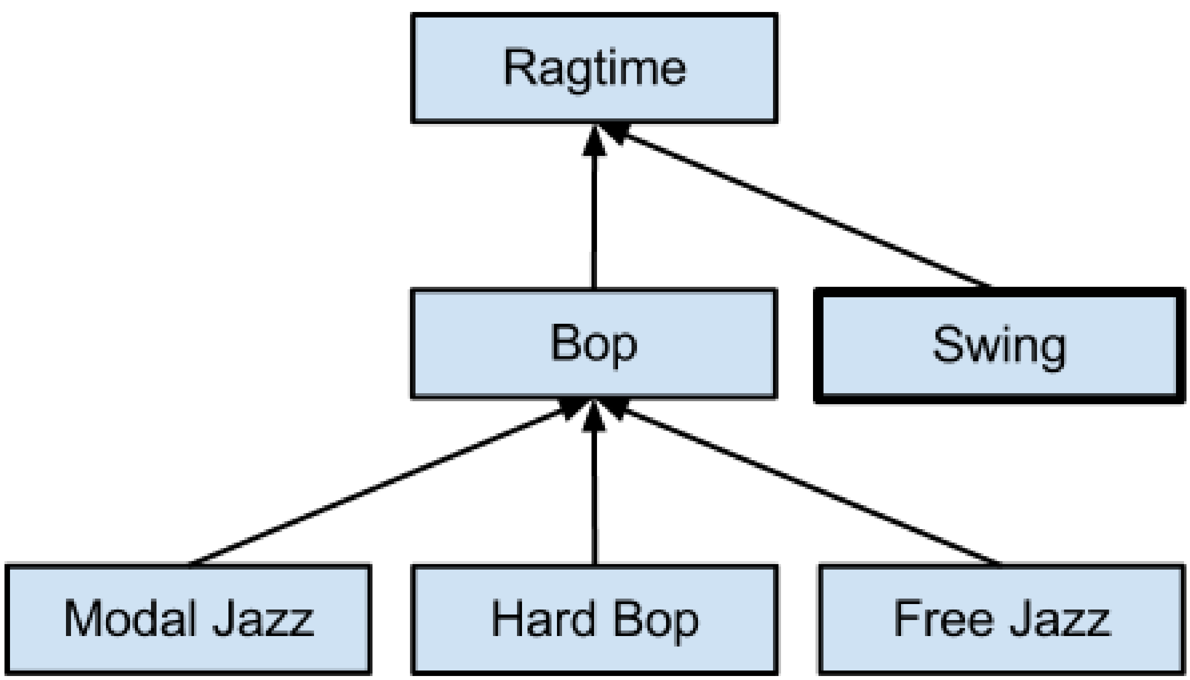


Figure 43.2: Adding a category

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 a 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:

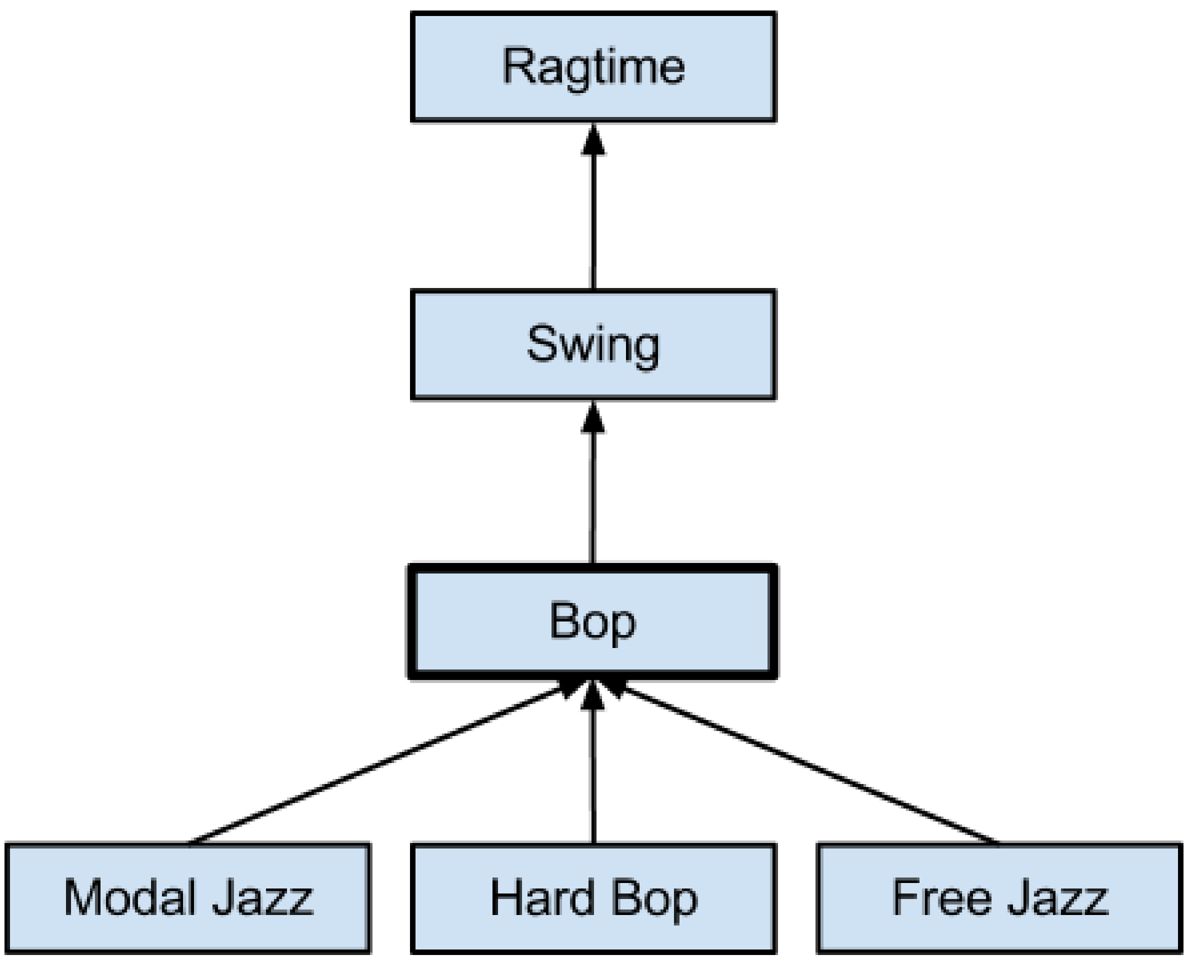


Figure 43.3: Change the parent of a category

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

[1] Your application cannot guarantee that the ancestor list of a parent category is correct, because MongoDB may process the categories out-of-order.

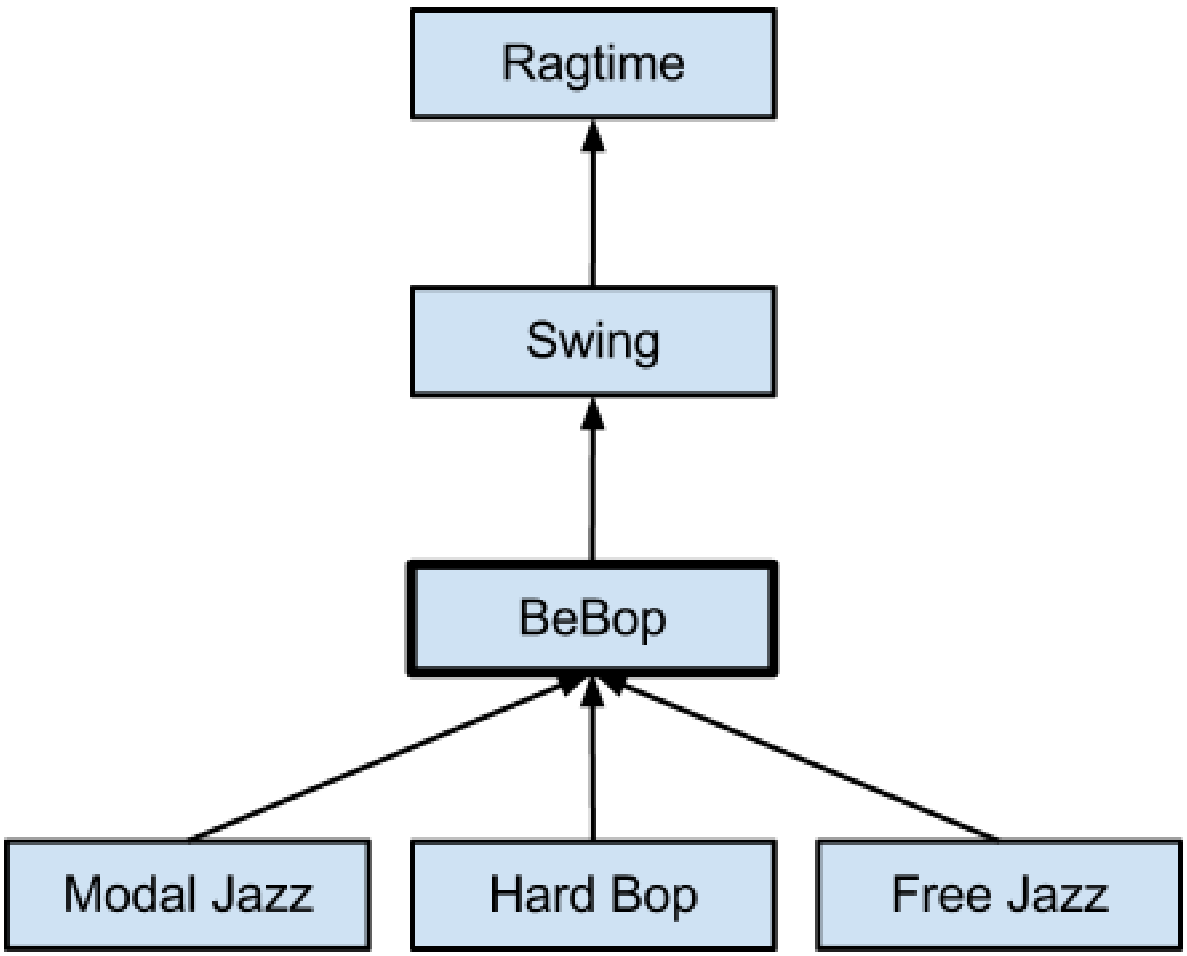


Figure 43.4: Rename a category

- 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.

43.3.3 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 44

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 681)" document introduces a model that you may use when designing a web site content management system, while "*Storing Comments* (page 688)" introduces the method for modeling user comments on content, like blog posts, and media, in MongoDB.

44.1 Metadata and Asset Management

44.1.1 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 types, 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.

44.1.2 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() } },
        w=1)
    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` (page 815) 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 a compound index on 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.

44.1.3 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 1061) 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.

44.2 Storing Comments

This document outlines the basic patterns for storing user-submitted comments in a content management system (CMS.)

44.2.1 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 1139), 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.

44.2.2 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 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 1140). 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` (page 985), `cursor.skip` (page 990), and `cursor.sort` (page 991)

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)])
```

44.2.3 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 1139). 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 (page 818) 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 (page 818) 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 819) 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 819) 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

`find_one()`.

44.2.4 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 1139), 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 `$push` (page 818) the comment onto the last page and `$inc` (page 810) 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 increments twice, that would 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 819) 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)])
```

44.2.5 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 1061) 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 45

Python Application Development

45.1 Write a Tumblelog Application with Django MongoDB Engine

45.1.1 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 3).

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.

45.1.2 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 an isolated Python environment for this project (named `myproject`.)

To activate `myproject` environment at the system prompt, use the following command:

```
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.

45.1.3 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

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'^$', ListView.as_view(
        queryset=Post.objects.all(),
        context_object_name="posts_list"),
        name="home"
    ),
    url(r'^post/(?P<slug>[a-zA-Z0-9-]+)/$', DetailView.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`:

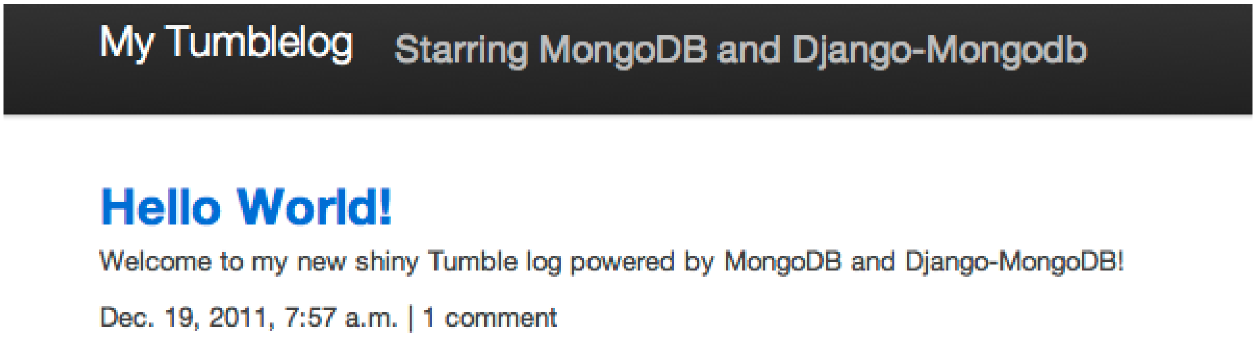
```
{% extends "base.html" %}

{% 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:



45.1.4 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

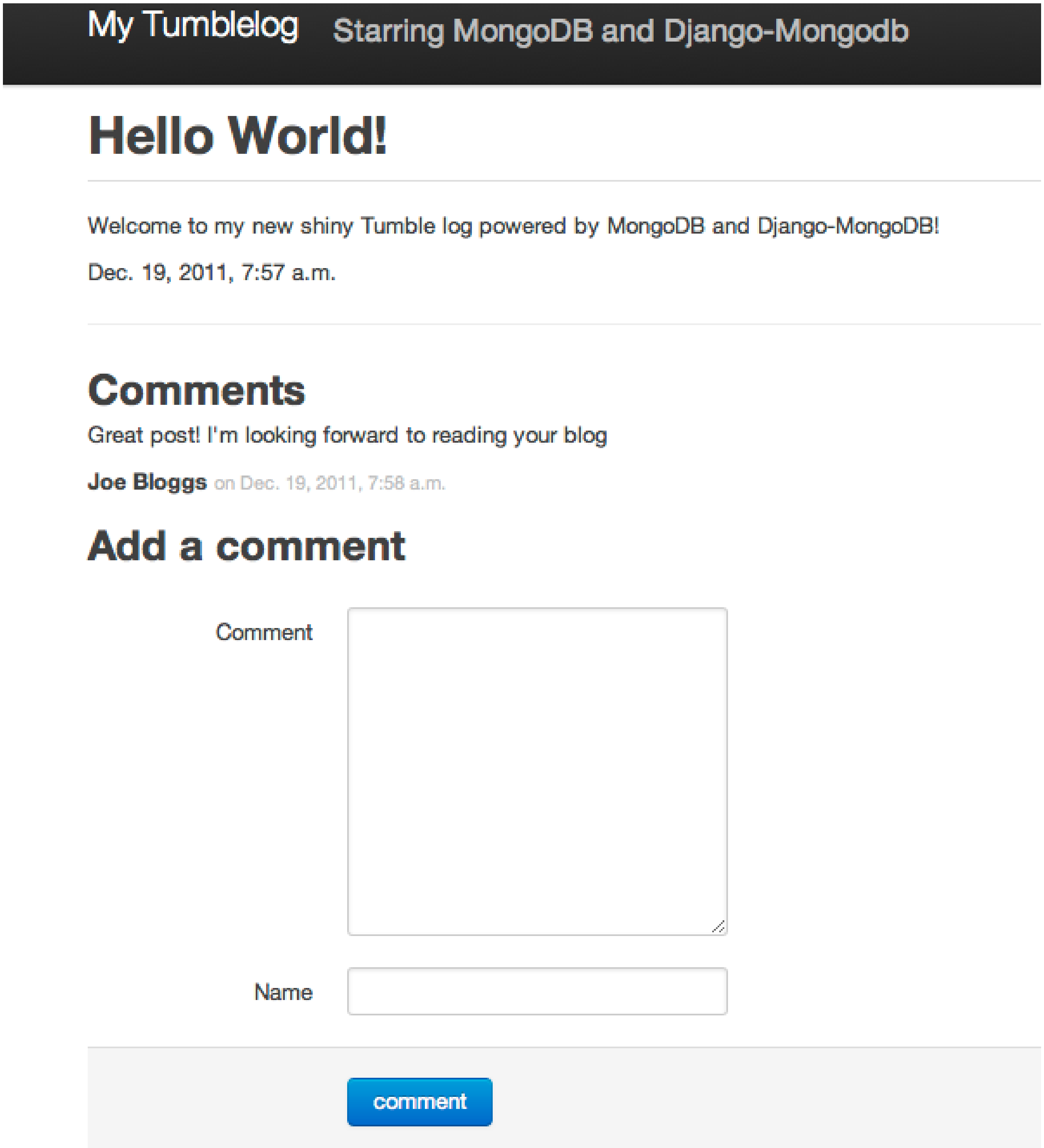
Finally, you can add the form to the templates, so that readers can create comments. Splitting the template for the forms out into `templates/_forms.html` will allow maximum reuse of forms code:

```
<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 by visiting http://localhost:8000/hello-world/. You should see the following:



45.1.5 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`
- `django_mongodb_engine`
- `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

The Django Admin documentation docs for additional information.

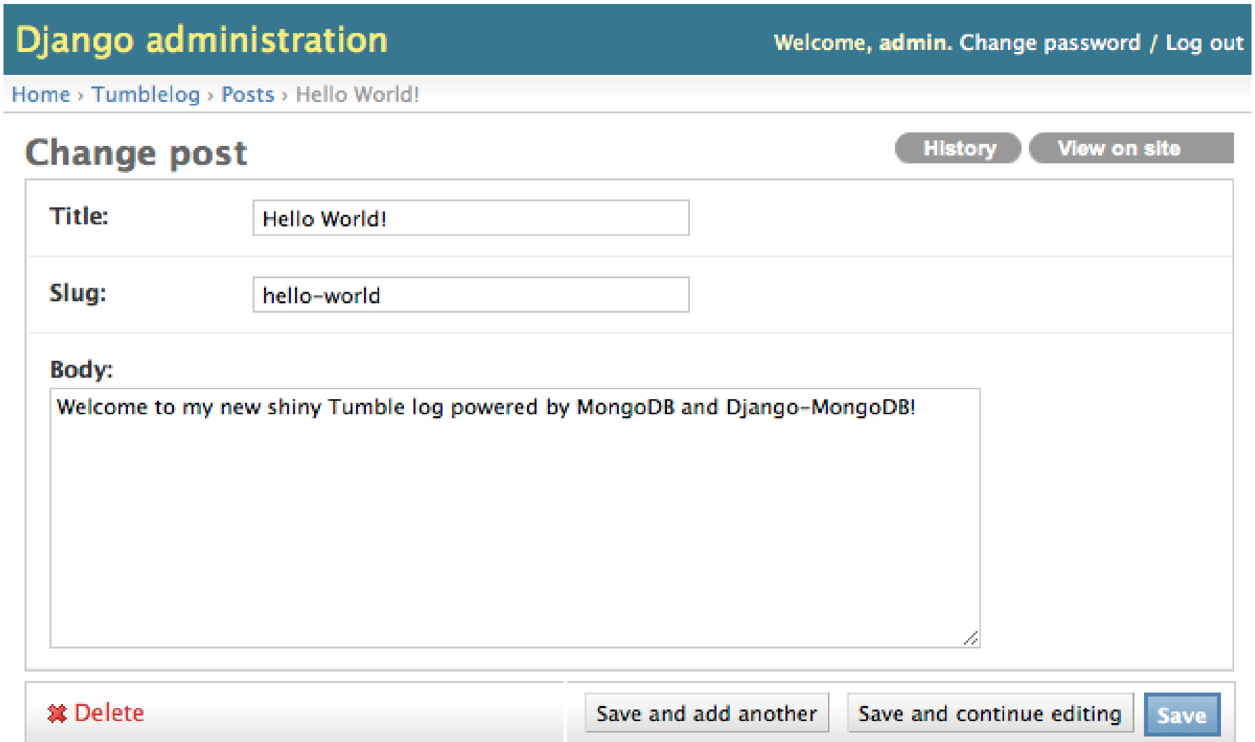
Update the urls.py to enable the administrative interface. Add the import and discovery mechanism to the top of the file and then add the admin import rule to the urlpatterns:

```
# 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/.



45.1.6 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.

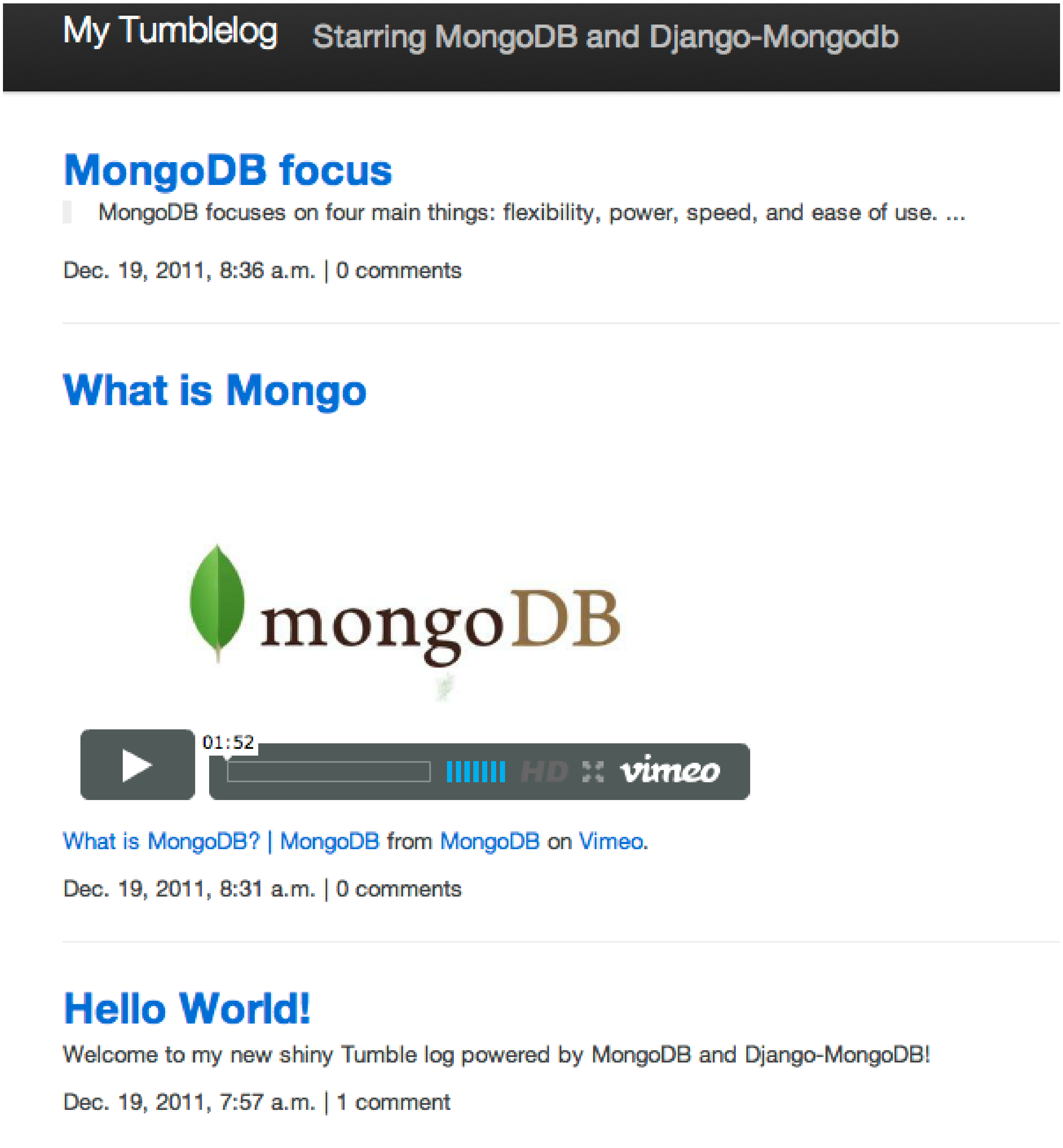
In the `post_list.html` file, change the post output display to resemble the following:

```
{% 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!



45.2 Write a Tumblelog Application with Flask and MongoEngine

45.2.1 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 3). 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.

45.2.2 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 an 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.

45.2.3 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_SETTINGS"] = {'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 MongoEng
          </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-%
      {{ 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:

45.2.4 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:

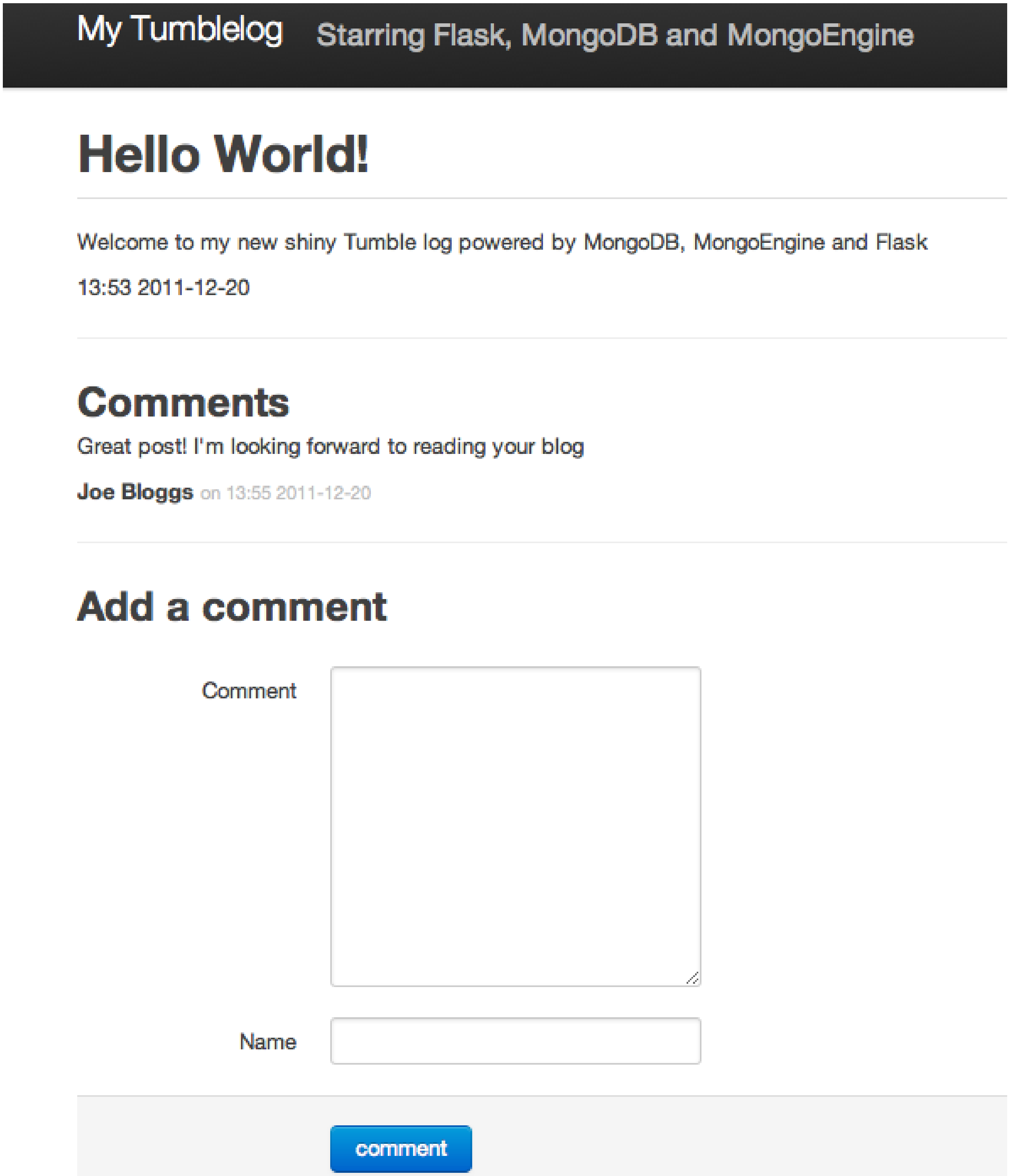
```
{% macro render(form) -%}
<fieldset>
{% for field in form %}
{% if field.type in ['CSRFTokenField', 'HiddenField'] %}
```

```
  {{ 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.



45.2.5 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 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 -%}
```

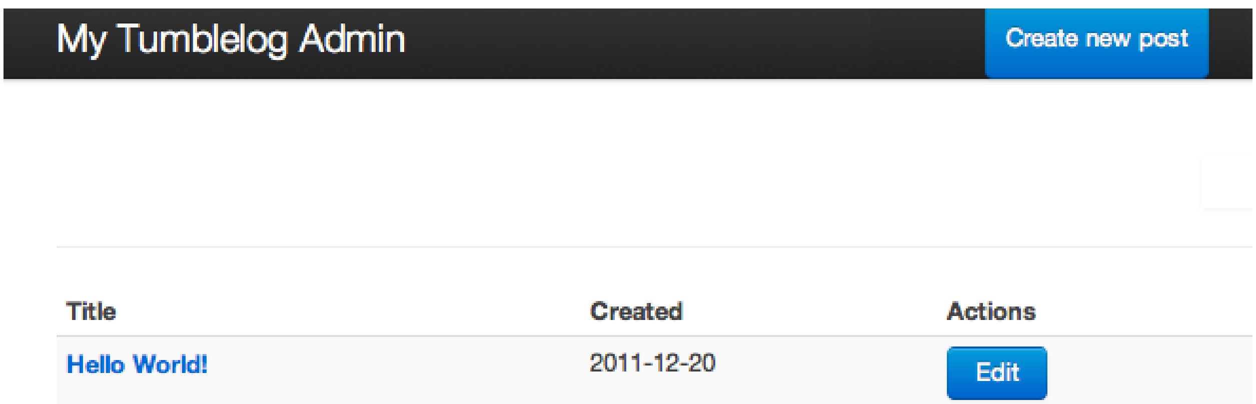
List all the posts in the `templates/admin/list.html` file:

```
{% 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 template 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`.)



45.2.6 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:

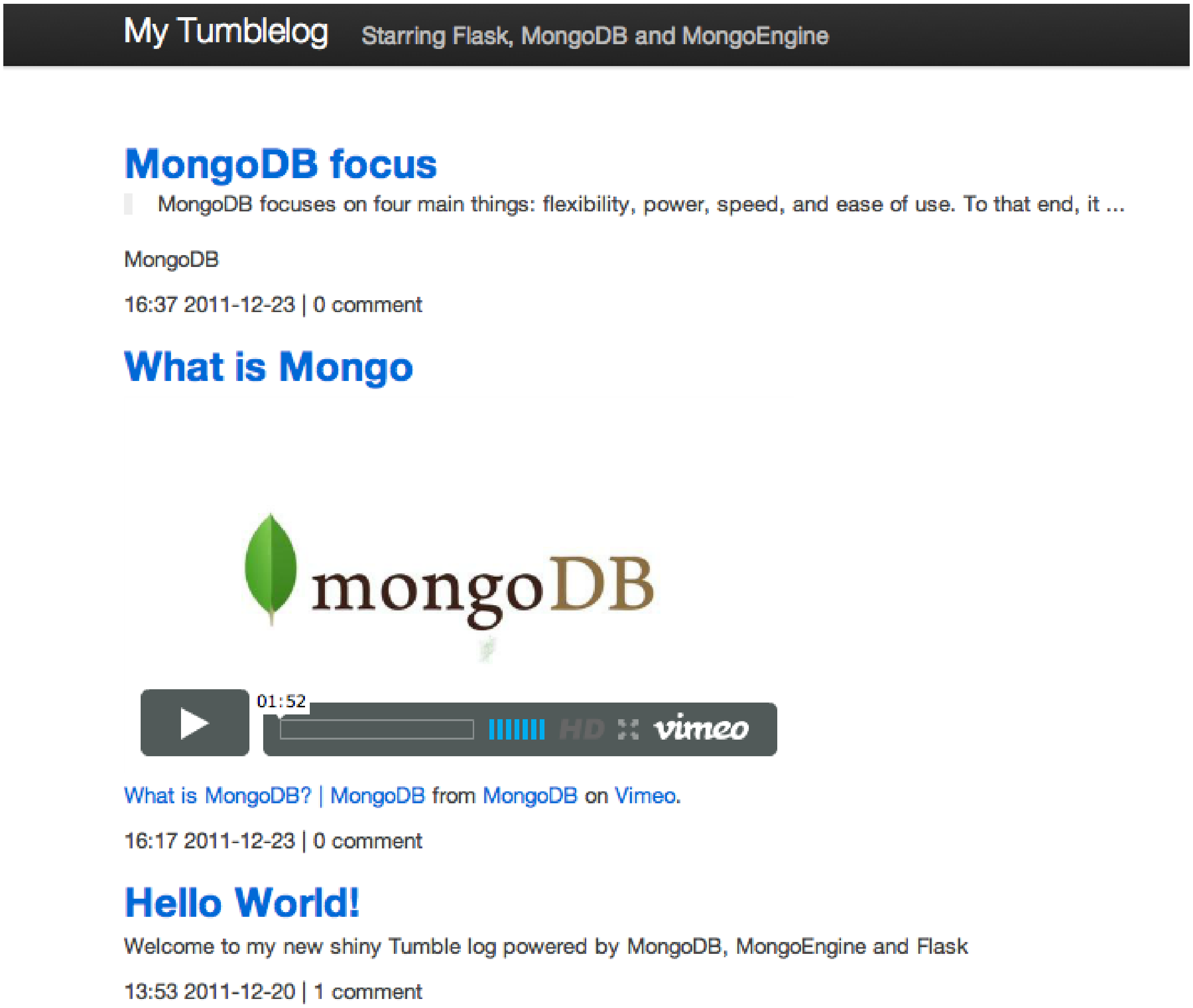
```
from tumblelog.models import Post, BlogPost, Video, Image, Quote, Comment

# ...

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!



45.2.7 Additional Resources

The complete source code is available on Github: <https://github.com/rozza/flask-tumblelog>

Part XIII

Frequently Asked Questions

CHAPTER 46

FAQ: MongoDB Fundamentals

Frequently Asked Questions:

This document addresses basic high level questions about MongoDB and its use.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 731) or post your question to the MongoDB User Mailing List.

46.1 What kind of database is MongoDB?

MongoDB is a *document*-oriented DBMS. Think of MySQL but with *JSON*-like objects comprising the data model, rather than RDBMS tables. Significantly, MongoDB supports neither joins nor transactions. However, it features secondary indexes, an expressive query language, atomic writes on a per-document level, and fully-consistent reads.

Operationally, MongoDB features master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

Note: MongoDB uses *BSON*, a binary object format similar to, but more expressive than *JSON*.

46.2 Do MongoDB databases have tables?

Instead of tables, a MongoDB database stores its data in *collections*, which are the rough equivalent of RDBMS tables. A collection holds one or more *documents*, which corresponds to a record or a row in a relational database table, and each document has one or more fields, which corresponds to a column in a relational database table.

Collections have important differences from RDBMS tables. Documents in a single collection may have a unique combination and set of fields. Documents need not have identical fields. You can add a field to some documents in a collection without adding that field to all documents in the collection.

See

SQL to MongoDB Mapping Chart (page 1042)

46.3 Do MongoDB databases have schemas?

MongoDB uses dynamic schemas. You can create collections without defining the structure, i.e. the fields or the types of their values, of the documents in the collection. You can change the structure of documents simply by adding new fields or deleting existing ones. Documents in a collection need not have an identical set of fields.

In practice, it is common for the documents in a collection to have a largely homogeneous structure; however, this is not a requirement. MongoDB's flexible schemas mean that schema migration and augmentation are very easy in practice, and you will rarely, if ever, need to write scripts that perform "alter table" type operations, which simplifies and facilitates iterative software development with MongoDB.

See

SQL to MongoDB Mapping Chart (page 1042)

46.4 What languages can I use to work with MongoDB?

MongoDB *client drivers* exist for all of the most popular programming languages, and many other ones. See the latest list of drivers for details.

See also:

"*MongoDB Drivers and Client Libraries* (page 575)."

46.5 Does MongoDB support SQL?

No.

However, MongoDB does support a rich, ad-hoc query language of its own.

See also:

Query, Update and Projection Operators (page 785)

46.6 What are typical uses for MongoDB?

MongoDB has a general-purpose design, making it appropriate for a large number of use cases. Examples include content management systems, mobile applications, gaming, e-commerce, analytics, archiving, and logging.

Do not use MongoDB for systems that require SQL, joins, and multi-object transactions.

46.7 Does MongoDB support transactions?

MongoDB does not provide ACID transactions.

However, MongoDB does provide some basic transactional capabilities. Atomic operations are possible within the scope of a single document: that is, we can debit `a` and credit `b` as a transaction if they are fields within the same document. Because documents can be rich, some documents contain thousands of fields, with support for testing fields in sub-documents.

Additionally, you can make writes in MongoDB durable (the 'D' in ACID). To get durable writes, you must enable journaling, which is on by default in 64-bit builds. You must also issue writes with a write concern of `{j: true}` to ensure that the writes block until the journal has synced to disk.

Users have built successful e-commerce systems using MongoDB, but applications requiring multi-object commits with rollback generally aren't feasible.

46.8 Does MongoDB require a lot of RAM?

Not necessarily. It's certainly possible to run MongoDB on a machine with a small amount of free RAM.

MongoDB automatically uses all free memory on the machine as its cache. System resource monitors show that MongoDB uses a lot of memory, but its usage is dynamic. If another process suddenly needs half the server's RAM, MongoDB will yield cached memory to the other process.

Technically, the operating system's virtual memory subsystem manages MongoDB's memory. This means that MongoDB will use as much free memory as it can, swapping to disk as needed. Deployments with enough memory to fit the application's working data set in RAM will achieve the best performance.

See also:

FAQ: MongoDB Diagnostics (page 777) for answers to additional questions about MongoDB and Memory use.

46.9 How do I configure the cache size?

MongoDB has no configurable cache. MongoDB uses all *free* memory on the system automatically by way of memory-mapped files. Operating systems use the same approach with their file system caches.

46.10 Does MongoDB require a separate caching layer for application-level caching?

No. In MongoDB, a document's representation in the database is similar to its representation in application memory. This means the database already stores the usable form of data, making the data usable in both the persistent store and in the application cache. This eliminates the need for a separate caching layer in the application.

This differs from relational databases, where caching data is more expensive. Relational databases must transform data into object representations that applications can read and must store the transformed data in a separate cache: if these transformation from data to application objects require joins, this process increases the overhead related to using the database which increases the importance of the caching layer.

46.11 Does MongoDB handle caching?

Yes. MongoDB keeps all of the most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

MongoDB does not implement a query cache: MongoDB serves all queries directly from the indexes and/or data files.

46.12 Are writes written to disk immediately, or lazily?

Writes are physically written to the *journal* (page 155) within 100 milliseconds, by default. At that point, the write is "durable" in the sense that after a pull-plug-from-wall event, the data will still be recoverable after a hard restart. See `journalCommitInterval` (page 1119) for more information on the journal commit window.

While the journal commit is nearly instant, MongoDB writes to the data files lazily. MongoDB may wait to write data to the data files for as much as one minute by default. This does not affect durability, as the journal has enough information to ensure crash recovery. To change the interval for writing to the data files, see `syncdelay` (page 1122).

46.13 What language is MongoDB written in?

MongoDB is implemented in C++. *Drivers* and client libraries are typically written in their respective languages, although some drivers use C extensions for better performance.

46.14 What are the limitations of 32-bit versions of MongoDB?

MongoDB uses *memory-mapped files* (page 767). 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 47

FAQ: MongoDB for Application Developers

Frequently Asked Questions:

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 731) or post your question to the MongoDB User Mailing List.

47.1 What is a namespace in MongoDB?

A "namespace" is the concatenation of the *database* name and the *collection* names [1] with a period character in between.

[1] Each index also has its own namespace.

Collections are containers for documents that share one or more indexes. Databases are groups of collections stored on disk using a single set of data files. [2]

For an example `acme.users` namespace, `acme` is the database name and `users` is the collection name. Period characters **can** occur in collection names, so that `acme.user.history` is a valid namespace, with `acme` as the database name, and `user.history` as the collection name.

While data models like this appear to support nested collections, the collection namespace is flat, and there is no difference from the perspective of MongoDB between `acme`, `acme.users`, and `acme.records`.

47.2 How do you copy all objects from one collection to another?

In the `mongo` (page 1066) 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 may incur a loss of type-fidelity.

Consider using `mongodump` (page 1075) and `mongorestore` (page 1079) to maintain type fidelity.

Also consider the `cloneCollection` (page 886) *command* that may provide some of this functionality.

47.3 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use `db.collection.remove()` (page 970), the object will no longer exist in MongoDB's on-disk data storage.

47.4 When does MongoDB write updates to disk?

MongoDB flushes writes to disk on a regular interval. In the default configuration, MongoDB writes data to the main data files on disk every 60 seconds and commits the *journal* roughly every 100 milliseconds. These values are configurable with the `journalCommitInterval` (page 1119) and `syncdelay` (page 1122).

These values represent the *maximum* amount of time between the completion of a write operation and the point when the write is durable in the journal, if enabled, and when MongoDB flushes data to the disk. In many cases MongoDB and the operating system flush data to disk more frequently, so that the above values represents a theoretical maximum.

However, by default, MongoDB uses a "lazy" strategy to write to disk. This is advantageous in situations where the database receives a thousand increments to an object within one second, MongoDB only needs to flush this data to disk once. In addition to the aforementioned configuration options, you can also use `fsync` (page 888) and `getLastError` (page 861) to modify this strategy.

47.5 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

[2] MongoDB database have an configurable limit on the `number of namespaces` (page 1139) in a database.

keeping transaction support extremely simple, MongoDB can provide greater performance especially for *partitioned* or *replicated* systems with a number of database server processes.

MongoDB *does* have support for atomic operations *within* a single document. Given the possibilities provided by nested documents, this feature provides support for a large number of use-cases.

See also:

The *Isolate Sequence of Operations* (page 593) page.

47.6 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new "*aggregation framework* (page 247)," with the `aggregate` (page 834) command.

MongoDB also supports *map-reduce* with the `mapReduce` (page 840) command, as well as basic aggregation with the `group` (page 836), `count` (page 834), and `distinct` (page 835). commands.

See also:

The *Aggregation* (page 245) page.

47.7 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 1123) option or the command-line option `--quiet` to suppress these messages from the log.

47.8 Does MongoDB run on Amazon EBS?

Yes.

MongoDB users of all sizes have had a great deal of success using MongoDB on the EC2 platform using EBS disks.

See also:

Amazon EC2

47.9 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 1122) setting to modify the file preallocation strategy.

See also:

Why are the files in my data directory larger than the data in my database? (page 769)

47.10 How do I optimize storage use for small documents?

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the `_id` field explicitly.

  MongoDB clients automatically add an `_id` field to each document and generate a unique 12-byte *ObjectId* for the `_id` field. Furthermore, MongoDB always indexes the `_id` field. For smaller documents this may account for a significant amount of space.

  To optimize storage use, users can specify a value for the `_id` field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the `_id` field that would have occupied space in another portion of the document.

  You can store any value in the `_id` field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field's value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.

  MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of documents that resemble the following:

  ```
  { 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.

47.11 When should I use GridFS?

For documents in a MongoDB collection, you should always use *GridFS* for storing files larger than 16 MB.

In some situations, storing large files may be more efficient in a MongoDB database than on a system-level filesystem.

- If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed.
- When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities. When using *geographically distributed replica sets* (page 388) MongoDB can distribute files and their metadata automatically to a number of `mongod` (page 1049) instances and facilities.
- When you want to access information from portions of large files without having to load whole files into memory, you can use GridFS to recall sections of files without reading the entire file into memory.

Do not use GridFS if you need to update the content of the entire file atomically. As an alternative you can store multiple versions of each file and specify the current version of the file in the metadata. You can update the metadata field that indicates "latest" status in an atomic update after uploading the new version of the file, and later remove previous versions if needed.

Furthermore, if your files are all smaller the 16 MB `BSON Document Size` (page 1139) limit, consider storing the file manually within a single document. You may use the BinData data type to store the binary data. See your *drivers* (page 575) documentation for details on using BinData.

For more information on GridFS, see *GridFS* (page 70).

47.12 How does MongoDB address SQL or Query injection?

47.12.1 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 575) 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.

47.12.2 JavaScript

Note: You can disable all server-side execution of JavaScript, by passing the *--noscripting* option on the command line or setting `noscripting` (page 1120) in a configuration file.

All of the following MongoDB operations permit you to run arbitrary JavaScript expressions directly on the server:

- `$where` (page 797)
- `db.eval()` (page 1002)
- `mapReduce` (page 840)
- `group` (page 836)

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` (page 797) field.

- If you need to pass user-supplied values in a `$where` (page 797) 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()` (page 1002) 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.

47.12.3 Dollar Sign Operator Escaping

Field names in MongoDB's query language have semantic meaning. The dollar sign (i.e `$`) is a reserved character used to represent *operators* (page 785) (i.e. `$inc` (page 810).) Thus, you should ensure that your application's users cannot inject operators into their inputs.

In some cases, you may wish to build a BSON object with a user-provided key. In these situations, keys will need to substitute the reserved `$` and `.` characters. Any character is sufficient, but consider using the Unicode full width equivalents: `U+FF04` (i.e. "$") and `U+FF0E` (i.e. ".").

Consider the following example:

```
BSONObj my_object = BSON( a_key << a_name );
```

The user may have supplied a `$` value in the `a_key` value. At the same time, `my_object` might be `{ $where : "things" }`. Consider the following cases:

- **Insert**. Inserting this into the database does no harm. The insert process does not evaluate the object as a query.

  Note: MongoDB client drivers, if properly implemented, check for reserved characters in keys on inserts.

- **Update**. The `db.collection.update()` (page 974) operation permits `$` operators in the update argument but does not support the `$where` (page 797) 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.

47.12.4 Driver-Specific Issues

See the "PHP MongoDB Driver Security Notes" page in the PHP driver documentation for more information

47.13 How does MongoDB provide concurrency?

MongoDB implements a readers-writer lock. This means that at any one time, only one client may be writing or any number of clients may be reading, but that reading and writing cannot occur simultaneously.

In standalone and *replica sets* the lock's scope applies to a single `mongod` (page 1049) instance or *primary* instance. In a sharded cluster, locks apply to each individual shard, not to the whole cluster.

For more information, see *FAQ: Concurrency* (page 749).

47.14 What is the compare order for BSON types?

MongoDB permits documents within a single collection to have fields with different *BSON* types. For instance, the following documents may exist within a single collection.

```
{ x: "string" }
{ x: 42 }
```

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Consider the following `mongo` (page 1066) 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 `$type` (page 795) operator provides access to *BSON type* comparison in the MongoDB query syntax. See the documentation on *BSON types* and the `$type` (page 795) operator for additional information.

Warning: Storing values of the different types in the same field in a collection is *strongly* discouraged.

See also:

- The *Tailable Cursors* (page 590) page for an example of a C++ use of `MinKey`.

47.15 How do I query for fields that have null values?

Fields in a document may store `null` values, as in a notional collection, `test`, with the following documents:

```
{ _id: 1, cancelDate: null }
{ _id: 2 }
```

Different query operators treat `null` values differently:

- The `{ cancelDate : null }` query matches documents that either contains the `cancelDate` field whose value is `null` *or* that do not contain the `cancelDate` field:

  ```
  db.test.find( { cancelDate: null } )
  ```

  The query returns both documents:

  ```
  { "_id" : 1, "cancelDate" : null }
  { "_id" : 2 }
  ```

- The `{ cancelDate : { $type: 10 } }` query matches documents that contains the `cancelDate` field whose value is `null` *only*; i.e. the value of the `cancelDate` field is of BSON Type `Null` (i.e. `10`) :

  ```
  db.test.find( { cancelDate : { $type: 10 } } )
  ```

  The query returns only the document that contains the `null` value:

  ```
  { "_id" : 1, "cancelDate" : null }
  ```

- The `{ cancelDate : { $exists: false } }` query matches documents that do not contain the `cancelDate` field:

  ```
  db.test.find( { cancelDate : { $exists: false } } )
  ```

  The query returns only the document that does *not* contain the `cancelDate` field:

  ```
  { "_id" : 2 }
  ```

See also:

The reference documentation for the `$type` (page 795) and `$exists` (page 794) operators.

47.16 Are there any restrictions on the names of Collections?

Collection names can be any UTF-8 string with the following exceptions:

- A collection name should begin with a letter or an underscore.
- The empty string (`""`) is not a valid collection name.
- Collection names cannot contain the `$` character. (version 2.2 only)
- Collection names cannot contain the null character: `\0`
- Do not name a collection using the `system.` prefix. MongoDB reserves `system.` for system collections, such as the `system.indexes` collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

If your collection name includes special characters, such as the underscore character, then to access the collection use the `db.getCollection()` (page 1005) 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()` (page 951) method, in as the following:

```
db.getCollection("_foo").find()
```

47.17 How do I isolate cursors from intervening write operations?

MongoDB cursors can return the same document more than once in some situations. [3] You can use the `snapshot()` (page 990) method on a cursor to isolate the operation for a very specific case.

`snapshot()` (page 990) traverses the index on the `_id` field and guarantees that the query will return each document (with respect to the value of the `_id` field) no more than once. [4]

The `snapshot()` (page 990) does not guarantee that the data returned by the query will reflect a single moment in time *nor* does it provide isolation from insert or delete operations.

Warning:

- You **cannot** use `snapshot()` (page 990) with *sharded collections*.
- You **cannot** use `snapshot()` (page 990) with `sort()` (page 991) or `hint()` (page 985) cursor methods.

As an alternative, if your collection has a field or fields that are never modified, you can use a *unique* index on this field or these fields to achieve a similar result as the `snapshot()` (page 990). Query with `hint()` (page 985) to explicitly force the query to use that index.

47.18 When should I embed documents within other documents?

When *modeling data in MongoDB* (page 107), embedding is frequently the choice for:

- "contains" relationships between entities.
- one-to-many relationships when the "many" objects *always* appear with or are viewed in the context of their parents.

You should also consider embedding for performance reasons if you have a collection with a large number of small documents. Nevertheless, if small, separate documents represent the natural model for the data, then you should maintain that model.

If, however, you can group these small documents by some logical relationship *and* you frequently retrieve the documents by this grouping, you might consider "rolling-up" the small documents into larger documents that contain an array of subdocuments. Keep in mind that if you often only need to retrieve a subset of the documents within the group, then "rolling-up" the documents may not provide better performance.

[3] As a cursor returns documents other operations may interleave with the query: if some of these operations are *updates* (page 93) that cause the document to move (in the case of a table scan, caused by document growth,) or that change the indexed field on the index used by the query; then the cursor will return the same document more than once.

[4] MongoDB does not permit changes to the value of the `_id` field; it is not possible for a cursor that transverses this index to pass the same document more than once.

"Rolling up" these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses.

Additionally, "rolling up" documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields *and* there would be fewer associated key entries in the corresponding index. See *Indexing Overview* (page 309) for more information on indexes.

47.19 Can I manually pad documents to prevent moves during updates?

An update can cause a document to move on disk if the document grows in size. To *minimize* document movements, MongoDB uses *padding* (page 55).

You should not have to pad manually because MongoDB adds *padding automatically* (page 55) and can adaptively adjust the amount of padding added to documents to prevent document relocations following updates.

You can change the default `paddingFactor` (page 901) calculation by using the `collMod` (page 892) command with the `usePowerOf2Sizes` (page 892) flag. The `usePowerOf2Sizes` (page 892) flag ensures that MongoDB allocates document space in sizes that are powers of 2, which helps ensure that MongoDB can efficiently reuse free pace created by document deletion or relocation.

However, in those exceptions where you must pad manually, you can use the strategy of first adding a temporary field to a document and then `$unset` (page 814) the field, as in the following example:

```
var myTempPadding = [ "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"];

db.myCollection.insert( { _id: 5, paddingField: myTempPadding } );

db.myCollection.update( { _id: 5 },
                        { $unset: { paddingField: "" } }
                      )

db.myCollection.update( { _id: 5 },
                        { $set: { realField: "Some text that I might have needed padding for" } }
                      )
```

See also:

Padding Factor (page 55)

CHAPTER 48

FAQ: The mongo Shell

Frequently Asked Questions:

48.1 How can I enter multi-line operations in the mongo shell?

If you end a line with an open parenthesis (`'('`), an open brace (`'{'`), or an open bracket (`'['`), then the subsequent lines start with ellipsis (`"..."`) until you enter the corresponding closing parenthesis (`')'`), the closing brace (`'}'`) or the closing bracket (`']'`). The `mongo` (page 1066) shell waits for the closing parenthesis, closing brace, or the closing bracket before evaluating the code, as in the following example:

```
> if ( x > 0 ) {
... count++;
... print (x);
... }
```

You can exit the line continuation mode if you enter two blank lines, as in the following example:

```
> if (x > 0
...
...
>
```

48.2 How can I access different databases temporarily?

You can use `db.getSiblingDB()` (page 1007) method to access another database without switching databases, as in the following example which first switches to the `test` database and then accesses the `sampleDB` database from the `test` database:

```
use test

db.getSiblingDB('sampleDB').getCollectionNames();
```

48.3 Does the mongo shell support tab completion and other keyboard shortcuts?

The mongo (page 1066) shell supports keyboard shortcuts. For example,

- Use the up/down arrow keys to scroll through command history. See *.dbshell* (page 1069) documentation for more information on the .dbshell file.
- Use <Tab> to autocomplete or to list the completion possibilities, as in the following example which uses <Tab> to complete the method name starting with the letter 'c':

  ```
  db.myCollection.c<Tab>
  ```

  Because there are many collection methods starting with the letter 'c', the <Tab> will list the various methods that start with 'c'.

For a full list of the shortcuts, see *Shell Keyboard Shortcuts* (page 1070)

48.4 How can I customize the mongo shell prompt?

New in version 1.9.

You can change the mongo (page 1066) shell prompt by setting the prompt variable. This makes it possible to display additional information in the prompt.

Set prompt to any string or arbitrary JavaScript code that returns a string, consider the following examples:

- Set the shell prompt to display the hostname and the database issued:

  ```
  var host = db.serverStatus().host;
  var prompt = function() { return db+"@"+host+"> "; }
  ```

  The mongo (page 1066) shell prompt should now reflect the new prompt:

  ```
  test@my-machine.local>
  ```

- Set the shell prompt to display the database statistics:

  ```
  var prompt = function() {
                  return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > "
               }
  ```

  The mongo (page 1066) shell prompt should now reflect the new prompt:

  ```
  Uptime:1052 Documents:25024787 >
  ```

You can add the logic for the prompt in the *.mongorc.js* (page 1069) file to set the prompt each time you start up the mongo (page 1066) shell.

48.5 Can I edit long shell operations with an external text editor?

You can use your own editor in the mongo (page 1066) shell by setting the EDITOR (page 1070) environment variable before starting the mongo (page 1066) shell. Once in the mongo (page 1066) shell, you can edit with the specified editor by typing edit <variable> or edit <function>, as in the following example:

1. Set the EDITOR (page 1070) variable from the command line prompt:

```
EDITOR=vim
```

2. Start the `mongo` (page 1066) shell:

```
mongo
```

3. Define a function `myFunction`:

```
function myFunction () { }
```

4. Edit the function using your editor:

```
edit myFunction
```

The command should open the `vim` edit session. Remember to save your changes.

5. Type `myFunction` to see the function definition:

```
myFunction
```

The result should be the changes from your saved edit:

```
function myFunction() {
    print("This was edited");
}
```

FAQ: Concurrency

Frequently Asked Questions:

Changed in version 2.2.

MongoDB allows multiple clients to read and write a single corpus of data using a locking system to ensure that all clients receive a consistent view of the data *and* to prevent multiple applications from modifying the exact same pieces of data at the same time. Locks help guarantee that all writes to a single document occur either in full or not at all.

See also:

Presentation on Concurrency and Internals in 2.2

49.1 What type of locking does MongoDB use?

MongoDB uses a readers-writer [1] lock that allows concurrent reads access to a database but gives exclusive access to a single write operation.

When a read lock exists, many read operations may use this lock. However, when a write lock exists, a single write operation holds the lock exclusively, and no other read *or* write operations may share the lock.

Locks are "writer greedy," which means writes have preference over reads. When both a read and write are waiting for a lock, MongoDB grants the lock to the write.

[1] You may be familiar with a "readers-writer" lock as "multi-reader" or "shared exclusive" lock. See the Wikipedia page on Readers-Writer Locks for more information.

49.2 How granular are locks in MongoDB?

Changed in version 2.2.

Beginning with version 2.2, MongoDB implements locks on a per-database basis for most read and write operations. Some global operations, typically short lived operations involving multiple databases, still require a global "instance" wide lock. Before 2.2, there is only one "global" lock per `mongod` (page 1049) instance.

For example, if you have six databases and one takes a write lock, the other five are still available for read and write.

49.3 How do I see the status of locks on my `mongod` instances?

For reporting on lock utilization information on locks, use any of the following methods:

- `db.serverStatus()` (page 1012),
- `db.currentOp()` (page 998),
- *mongotop* (page 1103),
- *mongostat* (page 1097), and/or
- the MongoDB Management Service (MMS)

Specifically, the `locks` (page 920) document in the *output of serverStatus* (page 919), or the `locks` (page 1000) field in the *current operation reporting* (page 998) provides insight into the type of locks and amount of lock contention in your `mongod` (page 1049) instance.

To terminate an operation, use `db.killOp()` (page 1009).

49.4 Does a read or write operation ever yield the lock?

New in version 2.0.

Read and write operations will yield their locks if the `mongod` (page 1049) receives a *page fault or* fetches data that is unlikely to be in memory. Yielding allows other operations that only need to access documents that are already in memory to complete while `mongod` (page 1049) loads documents into memory.

Additionally, write operations that affect multiple documents (i.e. `update()` (page 974) with the `multi` parameter,) will yield periodically to allow read operations during these long write operations. Similarly, long running read locks will yield periodically to ensure that write operations have the opportunity to complete.

Changed in version 2.2: The use of yielding expanded greatly in MongoDB 2.2. Including the "yield for page fault." MongoDB tracks the contents of memory and predicts whether data is available before performing a read. If MongoDB predicts that the data is not in memory a read operation yields its lock while MongoDB loads the data to memory. Once data is available in memory, the read will reacquire the lock to complete the operation.

49.5 Which operations lock the database?

Changed in version 2.2.

The following table lists common database operations and the types of locks they use.

Operation	Lock Type
Issue a query	Read lock
Get more data from a *cursor*	Read lock
Insert data	Write lock
Remove data	Write lock
Update data	Write lock
Map-reduce	Read lock and write lock, unless operations are specified as non-atomic. Portions of map-reduce jobs can run concurrently.
Create an index	Building an index in the foreground, which is the default, locks the database for extended periods of time.
`db.eval()` (page 1002)	Write lock. `db.eval()` (page 1002) blocks all other JavaScript processes.
`eval` (page 862)	Write lock. If used with the `nolock` lock option, the `eval` (page 862) option does not take a write lock and cannot write data to the database.
`aggregate()` (page 945)	Read lock

49.6 Which administrative commands lock the database?

Certain administrative commands can exclusively lock the database for extended periods of time. In some deployments, for large databases, you may consider taking the `mongod` (page 1049) instance offline so that clients are not affected. For example, if a `mongod` (page 1049) is part of a *replica set*, take the `mongod` (page 1049) offline and let other members of the set service load while maintenance is in progress.

The following administrative operations require an exclusive (i.e. write) lock on the database for extended periods:

- `db.collection.ensureIndex()` (page 949), when issued *without* setting `background` to `true`,
- `reIndex` (page 894),
- `compact` (page 890),
- `db.repairDatabase()` (page 1011),
- `db.createCollection()` (page 997), when creating a very large (i.e. many gigabytes) capped collection,
- `db.collection.validate()` (page 976), and
- `db.copyDatabase()` (page 996). This operation may lock all databases. See *Does a MongoDB operation ever lock more than one database?* (page 752).

The following administrative commands lock the database but only hold the lock for a very short time:

- `db.collection.dropIndex()`,
- `db.getLastError()` (page 1005),
- `db.isMaster()` (page 1008),
- `rs.status()` (page 1017) (i.e. `replSetGetStatus` (page 865),)
- `db.serverStatus()` (page 1012),
- `db.auth()` (page 995), and
- `db.addUser()` (page 993).

49.7 Does a MongoDB operation ever lock more than one database?

The following MongoDB operations lock multiple databases:

- `db.copyDatabase()` (page 996) must lock the entire `mongod` (page 1049) instance at once.
- *Journaling*, which is an internal operation, locks all databases for short intervals. All databases share a single journal.
- *User authentication* (page 211) locks the `admin` database as well as the database the user is accessing.
- All writes to a replica set's *primary* lock both the database receiving the writes and the `local` database. The lock for the `local` database allows the `mongod` (page 1049) to write to the primary's *oplog*.

49.8 How does sharding affect concurrency?

Sharding improves concurrency by distributing collections over multiple `mongod` (page 1049) instances, allowing shard servers (i.e. `mongos` (page 1061) processes) to perform any number of operations concurrently to the various downstream `mongod` (page 1049) instances.

Each `mongod` (page 1049) instance is independent of the others in the shard cluster and uses the MongoDB *readers-writer lock* (page 749)). The operations on one `mongod` (page 1049) instance do not block the operations on any others.

49.9 How does concurrency affect a replica set primary?

In *replication*, when MongoDB writes to a collection on the *primary*, MongoDB also writes to the primary's *oplog*, which is a special collection in the `local` database. Therefore, MongoDB must lock both the collection's database and the `local` database. The `mongod` (page 1049) must lock both databases at the same time keep both data consistent and ensure that write operations, even with replication, are "all-or-nothing" operations.

49.10 How does concurrency affect secondaries?

In *replication*, MongoDB does not apply writes serially to *secondaries*. Secondaries collect oplog entries in batches and then apply those batches in parallel. Secondaries do not allow reads while applying the write operations, and apply write operations in the order that they appear in the oplog.

MongoDB can apply several writes in parallel on replica set secondaries, in two phases:

1. During the first *prefer* phase, under a read lock, the `mongod` (page 1049) ensures that all documents affected by the operations are in memory. During this phase, other clients may execute queries against this member.
2. A thread pool using write locks applies all write operations in the batch as part of a coordinated write phase.

49.11 What kind of concurrency does MongoDB provide for JavaScript operations?

Changed in version 2.4: The V8 JavaScript engine added in 2.4 allows multiple JavaScript operations to run at the same time. Prior to 2.4, a single `mongod` (page 1049) could only run a *single* JavaScript operation at once.

FAQ: Sharding with MongoDB

Frequently Asked Questions:

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 731) or post your question to the MongoDB User Mailing List.

50.1 Is sharding appropriate for a new deployment?

Sometimes.

If your data set fits on a single server, you should begin with an unsharded deployment.

Converting an unsharded database to a *sharded cluster* is easy and seamless, so there is *little advantage* in configuring sharding while your data set is small.

Still, all production deployments should use *replica sets* to provide high availability and disaster recovery.

50.2 How does sharding work with replication?

To use replication with sharding, deploy each *shard* as a *replica set*.

50.3 Can I change the shard key after sharding a collection?

No.

There is no automatic support in MongoDB for changing a shard key after sharding a collection. This reality underscores the important of choosing a good *shard key* (page 502). 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 537) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See `shardCollection` (page 876), `sh.shardCollection()` (page 1027), the *Shard Key* (page 502), *Deploy a Sharded Cluster* (page 516), and SERVER-4000 for more information.

50.4 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.

50.5 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.

50.6 What happens if a client updates a document in a chunk during a migration?

The mongos (page 1061) routes the operation to the "old" shard, where it will succeed immediately. Then the *shard* mongod (page 1049) instances will replicate the modification to the "new" shard before the *sharded cluster* updates that chunk's "ownership," which effectively finalizes the migration process.

50.7 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 1061) will merely wait for the shard to return results.

50.8 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 1061) 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 1061) 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 1061) 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 1061) must parallelize the query over the shards and perform a merge-sort on the `email` of the documents found.

50.9 How does MongoDB sort queries in sharded environments?

If you call the `cursor.sort()` (page 991) method on a query in a sharded environment, the mongod (page 1049) for each shard will sort its results, and the mongos (page 1061) merges each shard's results before returning them to the client.

50.10 How does MongoDB ensure unique `_id` field values when using a shard key *other* than `_id`?

If you do not use `_id` as the shard key, then your application/client layer must be responsible for keeping the `_id` field unique. It is problematic for collections to have duplicate `_id` values.

If you're not sharding your collection by the `_id` field, then you should be sure to store a globally unique identifier in that field. The default *BSON ObjectID* (page 68) works well in this case.

50.11 I've enabled sharding and added a second shard, but all the data is still on one server. Why?

First, ensure that you've declared a *shard key* for your collection. Until you have configured the shard key, MongoDB will not create *chunks*, and *sharding* will not occur.

Next, keep in mind that the default chunk size is 64 MB. As a result, in most situations, the collection needs to have at least 64 MB of data before a migration will occur.

Additionally, the system which balances chunks among the servers attempts to avoid superfluous migrations. Depending on the number of shards, your shard key, and the amount of data, systems often require at least 10 chunks of data to trigger migrations.

You can run `db.printShardingStatus()` (page 1010) to see all the chunks present in your cluster.

50.12 Is it safe to remove old files in the `moveChunk` directory?

Yes. `mongod` (page 1049) creates these files as backups during normal *shard* balancing operations.

Once these migrations are complete, you may delete these files.

50.13 How does `mongos` use connections?

Each client maintains a connection to a `mongos` (page 1061) instance. Each `mongos` (page 1061) instance maintains a pool of connections to the members of a replica set supporting the sharded cluster. Clients use connections between `mongos` (page 1061) and `mongod` (page 1049) instances one at a time. Requests are not multiplexed or pipelined. When client requests complete, the `mongos` (page 1061) returns the connection to the pool.

See the *System Resource Utilization* (page 193) section of the *UNIX ulimit Settings* (page 193) document.

50.14 Why does `mongos` hold connections open?

`mongos` (page 1061) 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 1061) with a large number of open connections. If the `mongos` (page 1061) is no longer in use, it is safe to restart the process to close existing connections.

50.15 Where does MongoDB report on connections used by `mongos`?

Connect to the `mongos` (page 1061) with the `mongo` (page 1066) shell, and run the following command:

```
db._adminCommand("connPoolStats");
```

50.16 What does `writebacklisten` in the log mean?

The writeback listener is a process that opens a long poll to relay writes back from a `mongod` (page 1049) or `mongos` (page 1061) after migrations to make sure they have not gone to the wrong server. The writeback listener sends writes back to the correct server if necessary.

These messages are a key part of the sharding infrastructure and should not cause concern.

50.17 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.

50.18 What is the process for moving, renaming, or changing the number of config servers?

See *Sharded Cluster Tutorials* (page 515) for information on migrating and replacing config servers.

50.19 When do the `mongos` servers detect config server changes?

`mongos` (page 1061) 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 1061) 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` (page 873) command against any `mongos` (page 1061) to force it to refresh its cache.

50.20 Is it possible to quickly update `mongos` servers after updating a replica set configuration?

The `mongos` (page 1061) instances will detect these changes without intervention over time. However, if you want to force the `mongos` (page 1061) to reload its configuration, run the `flushRouterConfig` (page 873) command against to each `mongos` (page 1061) directly.

50.21 What does the maxConns setting on mongos do?

The maxConns (page 1116) option limits the number of connections accepted by mongos (page 1061).

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 1061) layer.

Set maxConns (page 1116) 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 1061) from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

50.22 How do indexes impact queries in sharded systems?

If the query does not include the *shard key*, the mongos (page 1061) 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 1061) can route the queries to a specific shard and the shard will use the index that will allow it to fulfill most efficiently. See this presentation for more information.

50.23 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.

50.24 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.

50.25 Can you shard on the _id field?

You can use any field for the shard key. The _id field is a common shard key.

Be aware that ObjectId() values, which are the default value of the _id field, increment as a timestamp. As a result, when used as a shard key, all new documents inserted into the collection will initially belong to the same chunk on a single shard. Although the system will eventually divide this chunk and migrate its contents to distribute data more evenly, at any moment the cluster can only direct insert operations at a single shard. This can limit the throughput of inserts. If most of your write operations are updates, this limitation should not impact your performance. However, if you have a high insert volume, this may be a limitation.

To address this issue, MongoDB 2.4 provides *hashed shard keys* (page 502).

50.26 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.

Changed in version 2.4: You can use a *hashed index* (page 315) and *hashed shard key* or you can compute and maintain this hashed value in your application.

50.27 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` (page 1049) issues this message if, during a *chunk migration* (page 511), 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` (page 880), 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.

50.28 How does draining a shard affect the balancing of uneven chunk distribution?

The sharded cluster balancing process controls both migrating chunks from decommissioned shards (i.e. draining,) and normal cluster balancing activities. Consider the following behaviors for different versions of MongoDB in situations where you remove a shard in a cluster with an uneven chunk distribution:

- After MongoDB 2.2, the balancer first removes the chunks from the draining shard and then balances the remaining uneven chunk distribution.
- Before MongoDB 2.2, the balancer handles the uneven chunk distribution and *then* removes the chunks from the draining shard.

CHAPTER 51

FAQ: Replica Sets and Replication in MongoDB

Frequently Asked Questions:

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 731) or post your question to the MongoDB User Mailing List.

51.1 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.

51.2 What do the terms "primary" and "master" mean?

Primary and *master* nodes are the nodes that can accept writes. MongoDB's replication is "single-master:" only one node can accept write operations at a time.

In a replica set, if the current "primary" node fails or becomes inaccessible, the other members can autonomously *elect* one of the other members of the set to be the new "primary".

By default, clients send all reads to the primary; however, *read preference* is configurable at the client level on a per-connection basis, which makes it possible to send reads to secondary nodes instead.

51.3 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.

51.4 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.

51.5 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 420)

51.6 Can MongoDB replicate over a "noisy" connection?

Yes, but not without connection failures and the obvious latency.

Members of the set will attempt to reconnect to the other members of the set in response to networking flaps. This does not require administrator intervention. However, if the network connections among the nodes in the replica set are very slow, it might not be possible for the members of the node to keep up with the replication.

If the TCP connection between the secondaries and the *primary* instance breaks, a *replica set* will automatically elect one of the *secondary* members of the set as primary.

51.7 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.

51.8 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.

51.9 Why use journaling if replication already provides data redundancy?

Journaling facilitates faster crash recovery. Prior to journaling, crashes often required `database repairs` (page 895) 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` (page 1049) 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.

51.10 Are write operations durable if write concern does not acknowledge writes?

Yes.

However, if you want confirmation that a given write has arrived at the server, use *write concern* (page 395). The `getLastError` (page 861) command provides the facility for write concern. However, after the *default write concern change* (page 1223), the default write concern acknowledges all write operations, and unacknowledged writes must be explicitly configured. See the *MongoDB Drivers and Client Libraries* (page 575) documentation for your driver for more information.

51.11 How many arbiters do replica sets need?

Some configurations do not require any *arbiter* instances. Arbiters vote in *elections* for *primary* but do not replicate the data like *secondary* members.

Replica sets require a majority of the remaining nodes present to elect a primary. Arbiters allow you to construct this majority without the overhead of adding replicating nodes to the system.

There are many possible replica set *architectures* (page 382).

If you have a three node replica set, you don't need an arbiter.

But a common configuration consists of two replicating nodes, one of which is *primary* and the other is *secondary*, as well as an arbiter for the third node. This configuration makes it possible for the set to elect a primary in the event of a failure without requiring three replicating nodes.

You may also consider adding an arbiter to a set if it has an equal number of nodes in two facilities and network partitions between the facilities are possible. In these cases, the arbiter will break the tie between the two facilities and allow the set to elect a new primary.

See also:

Replica Set Deployment Architectures (page 382)

51.12 What information do arbiters exchange with the rest of the replica set?

Arbiters never receive the contents of a collection but do exchange the following data with the rest of the replica set:

- Credentials used to authenticate the arbiter with the replica set. All MongoDB processes within a replica set use keyfiles. These exchanges are encrypted.
- Replica set configuration data and voting data. This information is not encrypted. Only credential exchanges are encrypted.

If your MongoDB deployment uses SSL, then all communications between arbiters and the other members of the replica set are secure. See the documentation for *Connect to MongoDB with SSL* (page 179) for more information. Run all arbiters on secure networks, as with all MongoDB components.

See

The overview of *Arbiter Members of Replica Sets* (page **??**).

51.13 Which members of a replica set vote in elections?

All members of a replica set, unless the value of `votes` (page 476) is equal to `0`, vote in elections. This includes all *delayed* (page 379), *hidden* (page 379) and *secondary-only* (page 378) members, as well as the *arbiters* (page **??**).

Additionally, the `state` (page 866) of the voting members also determine whether the member can vote. Only voting members in the following states are eligible to vote:

- `PRIMARY`
- `SECONDARY`
- `RECOVERING`
- `ARBITER`
- `ROLLBACK`

See also:

Replica Set Elections (page 389)

51.14 Do hidden members vote in replica set elections?

Hidden members (page 379) of *replica sets* *do* vote in elections. To exclude a member from voting in an *election*, change the value of the member's `votes` (page 476) configuration to `0`.

See also:

Replica Set Elections (page 389)

51.15 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 52

FAQ: MongoDB Storage

Frequently Asked Questions:

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 731) or post your question to the MongoDB User Mailing List.

52.1 What are memory mapped files?

A memory-mapped file is a file with data that the operating system places in memory by way of the `mmap()` system call. `mmap()` thus *maps* the file to a region of virtual memory. Memory-mapped files are the critical piece of the storage engine in MongoDB. By using memory mapped files MongoDB can treat the contents of its data files as if they were in memory. This provides MongoDB with an extremely fast and simple method for accessing and manipulating data.

52.2 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.

52.3 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.

52.4 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.

52.5 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.

52.6 What tools can I use to investigate storage use in MongoDB?

The `db.stats()` (page 1013) method in the `mongo` (page 1066) shell, returns the current state of the "active" database. The *dbStats command* (page 904) document describes the fields in the `db.stats()` (page 1013) output.

52.7 What is the working set?

Working set represents the total body of data that the application uses in the course of normal operation. Often this is a subset of the total data size, but the specific size of the working set depends on actual moment-to-moment use of the database.

If you run a query that requires MongoDB to scan every document in a collection, the working set will expand to include every document. Depending on physical memory size, this may cause documents in the working set to "page out," or to be removed from physical memory by the operating system. The next time MongoDB needs to access these documents, MongoDB may incur a hard page fault.

If you run a query that requires MongoDB to scan every *document* in a collection, the working set includes every active document in memory.

For best performance, the majority of your *active* set should fit in RAM.

52.8 Why are the files in my data directory larger than the data in my database?

The data files in your data directory, which is the `http://docs.mongodb.org/manualdata/db` directory in default configurations, might be larger than the data set inserted into the database. Consider the following possible causes:

- Preallocated data files.

  In the data directory, MongoDB preallocates data files to a particular size, in part to prevent file system fragmentation. MongoDB names the first data file `<databasename>.0`, the next `<databasename>.1`, etc. The first file `mongod` (page 1049) allocates is 64 megabytes, the next 128 megabytes, and so on, up to 2 gigabytes, at which point all subsequent files are 2 gigabytes. The data files include files with allocated space but that hold no data. `mongod` (page 1049) may allocate a 1 gigabyte data file that may be 90% empty. For most larger databases, unused allocated space is small compared to the database.

  On Unix-like systems, `mongod` (page 1049) preallocates an additional data file and initializes the disk space to `0`. Preallocating data files in the background prevents significant delays when a new database file is next allocated.

  You can disable preallocation with the `noprealloc` (page 1120) run time option. However `noprealloc` (page 1120) is **not** intended for use in production environments: only use `noprealloc` (page 1120) for testing and with small data sets where you frequently drop databases.

  On Linux systems you can use `hdparm` to get an idea of how costly allocation might be:

  ```
  time hdparm --fallocate $((1024*1024)) testfile
  ```

- The *oplog*.

  If this `mongod` (page 1049) is a member of a replica set, the data directory includes the *oplog.rs* file, which is a preallocated *capped collection* in the `local` database. The default allocation is approximately 5% of disk space on 64-bit installations, see *Oplog Sizing* (page 405) for more information. In most cases, you should not need to resize the oplog. However, if you do, see *Change the Size of the Oplog* (page 439).

- The *journal*.

  The data directory contains the journal files, which store write operations on disk prior to MongoDB applying them to databases. See *Journaling* (page 155).

- Empty records.

  MongoDB maintains lists of empty records in data files when deleting documents and collections. MongoDB can reuse this space, but will never return this space to the operating system.

  To reclaim deleted space, use either of the following:

  - `compact` (page 890), which de-fragments deleted space. `compact` (page 890) requires up to 2 gigabytes of extra disk space to run. Do not use `compact` (page 890) if you are critically low on disk space.

    Important: `compact` (page 890) only removes fragmentation from MongoDB data files and does not return any disk space to the operating system.

  - `repairDatabase` (page 895), which rebuilds the database. Both options require additional disk space to run. For details, see *Recover MongoDB Data following Unexpected Shutdown* (page 144).

    Warning: `repairDatabase` (page 895) requires enough free disk space to hold both the old and new database files while the repair is running. Be aware that `repairDatabase` (page 895) will block all other operations and may take a long time to complete.

52.9 How can I check the size of a collection?

To view the size of a collection and other information, use the `stats()` (page 973) method from the `mongo` (page 1066) shell. The following example issues `stats()` (page 973) for the `orders` collection:

```
db.orders.stats();
```

To view specific measures of size, use these methods:

- `db.collection.dataSize()` (page 948): data size for the collection.
- `db.collection.storageSize()` (page 973): allocation size, including unused space.
- `db.collection.totalSize()` (page 973): the data size plus the index size.
- `db.collection.totalIndexSize()` (page 973): the index size.

Also, the following scripts print the statistics for each database and collection:

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); prin
```

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); mdb
```

52.10 How can I check the size of indexes?

To view the size of the data allocated for an index, use one of the following procedures in the `mongo` (page 1066) shell:

- Use the `stats()` (page 973) method using the index namespace. To retrieve a list of namespaces, issue the following command:

  ```
  db.system.namespaces.find()
  ```

- Check the value of `indexSizes` (page 902) in the output of the `db.collection.stats()` (page 973) command.

Example

Issue the following command to retrieve index namespaces:

```
db.system.namespaces.find()
```

The command returns a list similar to the following:

```
{"name" : "test.orders"}
{"name" : "test.system.indexes"}
{"name" : "test.orders.$_id_"}
```

View the size of the data allocated for the `orders.$_id_` index with the following sequence of operations:

```
use test
db.orders.$_id_.stats().indexSizes
```

52.11 How do I know when the server runs out of disk space?

If your server runs out of disk space for data files, you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:09 [FileAllocator]     will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:19 [FileAllocator]     will try again in 10 seconds
```

The server remains in this state forever, blocking all writes including deletes. However, reads still work. To delete some data and compact, using the `compact` (page 890) command, you must restart the server first.

If your server runs out of disk space for journal files, the server process will exit. By default, `mongod` (page 1049) creates journal files in a sub-directory of `dbpath` (page 1118) named `journal`. You may elect to put the journal files on another storage device using a filesystem mount or a symlink.

Note: If you place the journal files on a separate storage device you will not be able to use a file system snapshot tool to capture a consistent snapshot of your data files and journal files.

FAQ: Indexes

Frequently Asked Questions:

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 731) or post your question to the MongoDB User Mailing List. See also *Indexing Strategies* (page 321).

53.1 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()` (page 949) is usually ok, if an index doesn't exist because of ongoing administrative work, a call to `ensureIndex()` (page 949) may disrupt database availability. Running `ensureIndex()` (page 949) can render a replica set inaccessible as the index creation is happening. See *Build Indexes on Replica Sets* (page 333).

53.2 How do you know what indexes exist in a collection?

To list a collection's indexes, use the `db.collection.getIndexes()` (page 956) method or a similar method for your driver.

53.3 How do you determine the size of an index?

To check the sizes of the indexes on a collection, use `db.collection.stats()` (page 973).

53.4 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 326).

53.5 How do you know what index a query used?

To inspect how MongoDB processes a query, use the `explain()` (page 979) method in the `mongo` (page 1066) shell, or in your application driver.

53.6 How do you determine what fields to index?

A number of factors determine what fields to index, including *selectivity* (page 326), 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 321).

53.7 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.

53.8 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 315).

If you build a large index without the background option, and if doing so causes the database to stop responding, do one of the following:

- Wait for the index to finish building.
- Kill the current operation (see `db.killOp()` (page 1009)). The partial index will be deleted.

53.9 Can I use index keys to constrain query matches?

You can use the `min()` (page 987) and `max()` (page 986) methods to constrain the results of the cursor returned from `find()` (page 951) by using index keys.

53.10 Using `$ne` and `$nin` in a query is slow. Why?

The `$ne` (page 789) and `$nin` (page 789) operators are not selective. See *Create Queries that Ensure Selectivity* (page 326). If you need to use these, it is often best to make sure that an additional, more selective criterion is part of the query.

53.11 Can I use a multi-key index to support a query for a whole array?

Not entirely. The index can partially support these queries because it can speed the selection of the first element of the array; however, comparing all subsequent items in the array cannot use the index and must scan the documents individually.

53.12 How can I effectively use indexes strategy for attribute lookups?

For simple attribute lookups that don't require sorted result sets or range queries, consider creating a field that contains an array of documents where each document has a field (e.g. `attrib`) that holds a specific type of attribute. You can index this `attrib` field.

For example, the `attrib` field in the following document allows you to add an unlimited number of attributes types:

```
{ _id : ObjectId(...),
  attrib : [
            { k: "color", v: "red" },
            { k: "shape": v: "rectangle" },
            { k: "color": v: "blue" },
            { k: "avail": v: true }
           ]
}
```

Both of the following queries could use the same `{ "attrib.k": 1, "attrib.v": 1 }` index:

```
db.mycollection.find( { attrib: { $elemMatch : { k: "color", v: "blue" } } } )
db.mycollection.find( { attrib: { $elemMatch : { k: "avail", v: true } } } )
```

FAQ: MongoDB Diagnostics

Frequently Asked Questions:

This document provides answers to common diagnostic questions and issues.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 731) or post your question to the MongoDB User Mailing List.

54.1 Where can I find information about a `mongod` process that stopped running unexpectedly?

If `mongod` (page 1049) shuts down unexpectedly on a UNIX or UNIX-based platform, and if `mongod` (page 1049) fails to log a shutdown or error message, then check your system logs for messages pertaining to MongoDB. For example, for logs located in `http://docs.mongodb.org/manualvar/log/messages`, use the following commands:

```
sudo grep mongod /var/log/messages
sudo grep score /var/log/messages
```

54.2 Does TCP `keepalive` time affect sharded clusters and replica sets?

If you experience socket errors between members of a sharded cluster or replica set, that do not have other reasonable causes, check the TCP keep alive value, which Linux systems store as the `tcp_keepalive_time` value. A common keep alive period is `7200` seconds (2 hours); however, different distributions and OS X may have different settings. For MongoDB, you will have better experiences with shorter keepalive periods, on the order of `300` seconds (five minutes).

On Linux systems you can use the following operation to check the value of `tcp_keepalive_time`:

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

You can change the `tcp_keepalive_time` value with the following operation:

```
echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time
```

The new `tcp_keepalive_time` value takes effect without requiring you to restart the `mongod` (page 1049) or `mongos` (page 1061) servers. When you reboot or restart your system you will need to set the new `tcp_keepalive_time` value, or see your operating system's documentation for setting the TCP keepalive value persistently.

For OS X systems, issue the following command to view the keep alive setting:

```
sysctl net.inet.tcp.keepinit
```

To set a shorter keep alive period use the following invocation:

```
sysctl -w net.inet.tcp.keepinit=300
```

If your replica set or sharded cluster experiences keepalive-related issues, you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This includes all machines hosting `mongos` (page 1061) or `mongod` (page 1049) servers.

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

54.3 Memory Diagnostics

54.3.1 Do I need to configure swap space?

Always configure systems to have swap space. Without swap, your system may not be reliant in some situations with extreme memory constraints, memory leaks, or multiple programs using the same memory. Think of the swap space as something like a steam release valve that allows the system to release extra pressure without affecting the overall functioning of the system.

Nevertheless, systems running MongoDB *do not* need swap for routine operation. Database files are *memory-mapped* (page 767) and should constitute most of your MongoDB memory use. Therefore, it is unlikely that `mongod` (page 1049) will ever use any swap space in normal operation. The operating system will release memory from the memory mapped files without needing swap and MongoDB can write data to the data files without needing the swap system.

54.3.2 What is "working set" and how can I estimate it's size?

The *working set* for a MongoDB database is the portion of your data that clients access most often. You can estimate size of the working set, using the `workingSet` (page 932) document in the output of `serverStatus` (page 919). To return `serverStatus` (page 919) with the `workingSet` (page 932) document, issue a command in the following form:

```
db.runCommand( { serverStatus: 1, workingSet: 1 } )
```

54.3.3 Must my working set size fit RAM?

Your working set should stay in memory to achieve good performance. Otherwise many random disk IO's will occur, and unless you are using SSD, this can be quite slow.

One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (as would happen with id's that are randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id's in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

It is fine if databases and thus virtual size are much larger than RAM.

54.3.4 How do I calculate how much RAM I need for my application?

The amount of RAM you need depends on several factors, including but not limited to:

- The relationship between *database storage* (page 767) and working set.
- The operating system's cache strategy for LRU (Least Recently Used)
- The impact of *journaling* (page 155)
- The number or rate of page faults and other MMS gauges to detect when you need more RAM

MongoDB defers to the operating system when loading data into memory from disk. It simply *memory maps* (page 767) all its data files and relies on the operating system to cache data. The OS typically evicts the least-recently-used data from RAM when it runs low on memory. For example if clients access indexes more frequently than documents, then indexes will more likely stay in RAM, but it depends on your particular usage.

To calculate how much RAM you need, you must calculate your working set size, or the portion of your data that clients use most often. This depends on your access patterns, what indexes you have, and the size of your documents.

If page faults are infrequent, your working set fits in RAM. If fault rates rise higher than that, you risk performance degradation. This is less critical with SSD drives than with spinning disks.

54.3.5 How do I read memory statistics in the UNIX `top` command

Because `mongod` (page 1049) uses *memory-mapped files* (page 767), the memory statistics in `top` require interpretation in a special way. On a large database, `VSIZE` (virtual bytes) tends to be the size of the entire database. If the `mongod` (page 1049) doesn't have other processes running, `RSIZE` (resident bytes) is the total memory of the machine, as this counts file system cache contents.

For Linux systems, use the `vmstat` command to help determine how the system uses memory. On OS X systems use `vm_stat`.

54.4 Sharded Cluster Diagnostics

The two most important factors in maintaining a successful sharded cluster are:

- *choosing an appropriate shard key* (page 502) and
- *sufficient capacity to support current and future operations* (page 499).

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.

54.4.1 In a new sharded cluster, why does all data remains on one shard?

Your cluster must have sufficient data for sharding to make sense. Sharding works by migrating chunks between the shards until each shard has roughly the same number of chunks.

The default chunk size is 64 megabytes. MongoDB will not begin migrations until the imbalance of chunks in the cluster exceeds the *migration threshold* (page 510). While the default chunk size is configurable with the `chunkSize` (page 1126) 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 512) 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 536).

Finally, if your shard key has a low *cardinality* (page 521), MongoDB may not be able to create sufficient splits among the data.

54.4.2 Why would one shard receive a disproportion amount of traffic in a sharded cluster?

In some situations, a single shard or a subset of the cluster will receive a disproportionate portion of the traffic and workload. In almost all cases this is the result of a shard key that does not effectively allow *write scaling* (page 503).

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 519) to correct this pattern.

54.4.3 What can prevent a sharded cluster from balancing?

If you have just deployed your sharded cluster, you may want to consider the *troubleshooting suggestions for a new cluster where data remains on a single shard* (page 780).

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 521) 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 542) that is too short, given the rate of data growth.
  - an uneven distribution of *write operations* (page 503) 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.

54.4.4 Why do chunk migrations affect sharded cluster performance?

If migrations impact your cluster or application's performance, consider the following options, depending on the nature of the impact:

1. If migrations only interrupt your clusters sporadically, you can limit the *balancing window* (page 542) 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 538) 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 523) 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 503).

Part XIV

Reference

CHAPTER 55

MongoDB Interface

55.1 Query, Update and Projection Operators

55.1.1 Query Selectors

Comparison

Comparison Query Operators

Name	Description
`$all` (page 785)	Matches arrays that contain all elements specified in the query.
`$gt` (page 786)	Matches values that are greater than the value specified in the query.
`$gte` (page 787)	Matches values that are equal to or greater than the value specified in the query.
`$in` (page 787)	Matches any of the values that exist in an array specified in the query.
`$lt` (page 788)	Matches values that are less than the value specified in the query.
`$lte` (page 788)	Matches values that are less than or equal to the value specified in the query.
`$ne` (page 789)	Matches all values that are not equal to the value specified in the query.
`$nin` (page 789)	Matches values that **do not** exist in an array specified to the query.

$all

$all

Syntax: `{ field: { $all: [ <value> , <value1> ... ] }`

`$all` (page 785) 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 `book`.

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 `$all` (page 785) operator exists to describe and specify arrays in MongoDB queries. However, you may use the `$all` (page 785) 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` (page 785) 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()` (page 951), `update()` (page 974), and `$set` (page 814).

$gt

$gt

Syntax: `{field: {$gt: value} }`

`$gt` (page 786) 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` (page 786) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 974) 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()` (page 951), `update()` (page 974), `$set` (page 814).

$gte

`$gte`

Syntax: `{field: {$gte: value} }`

`$gte` (page 787) 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` (page 787) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 974) 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()` (page 951), `update()` (page 974), `$set` (page 814).

$in

`$in`

Syntax: `{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }`

`$in` (page 787) 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 selects all documents in the `inventory` collection where the `qty` field value is either `5` or `15`. Although you can express this query using the `$or` (page 790) operator, choose the `$in` (page 787) operator rather than the `$or` (page 790) operator when performing equality checks on the same field.

If the `field` holds an array, then the `$in` (page 787) 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()` (page 974) 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"]`.

Note: When using two or more `$in` (page 787) expressions, the product of the number of **distinct** elements in the `$in` (page 787) arrays must be less than 4000000. Otherwise, MongoDB will throw an exception of `"combinatorial limit of $in partitioning of result set exceeded"`.

See also:

`find()` (page 951), `update()` (page 974), `$or` (page 790), `$set` (page 814).

$lt

`$lt`

Syntax: `{field: {$lt: value} }`

`$lt` (page 788) 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` (page 788) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lt: 20 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 974) 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()` (page 951), `update()` (page 974), `$set` (page 814).

$lte

`$lte`

Syntax: `{ field: { $lte: value} }`

`$lte` (page 788) 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` (page 788) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.fee": { $lte: 5 } }, { $set: { price: 9.99 } } )
```

This `update()` (page 974) 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()` (page 951), `update()` (page 974), `$set` (page 814).

$ne

$ne

Syntax: {field: {$ne: value} }

$ne (page 789) 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 (page 789) operator with a field from an embedded document:

```
db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )
```

This update() (page 974) 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() (page 951), update() (page 974), $set (page 814).

$nin

$nin

Syntax: { field: { $nin: [<value1>, <value2> ... <valueN>]} }

$nin (page 789) 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 (page 789) 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() (page 974) 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() (page 951), update() (page 974), $set (page 814).

Logical

Logical Query Operators

Name	Description
$or (page 790)	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and (page 791)	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not (page 792)	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$nor (page 793)	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.

$or

$or

New in version 1.6.

Changed in version 2.0: You may nest $or (page 790) operations; however, these expressions are not as efficiently optimized as top-level.

Syntax: { $or: [{ <expression1> }, { <expression2> }, ... , { <expressionN> }] }

The $or (page 790) 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 (page 790) operator to select fields from embedded documents:

```
db.inventory.update( { $or: [ { price:10.99 }, { "carrier.state": "NY"} ] }, { $set: { sale: tru
```

This update() (page 974) 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 (page 790) with <expressions> that are equality checks for the value of the same field, choose the $in (page 787) operator over the $or (page 790) 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` (page 790) operator:

- When using indexes with `$or` (page 790) queries, remember that each clause of an `$or` (page 790) 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 `$or` (page 790) operator with the `sort()` (page 991) method in a query, the query will **not** use the indexes on the `$or` (page 790) fields. Consider the following query which adds a `sort()` (page 991) 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`.

- You cannot use the `$or` (page 790) with `2d` *geospatial queries* (page 351).

See also:

`find()` (page 951), `update()` (page 974), `$set` (page 814), `$and` (page 791), `sort()` (page 991).

$and

`$and`

New in version 2.0.

Syntax: `{ $and: [ { <expression1> }, { <expression2> } , ... , { <expressionN> } ] }`

`$and` (page 791) performs a logical `AND` operation on an array of *two or more* expressions (e.g. `<expression1>`, `<expression2>`, etc.) and selects the documents that satisfy *all* the expressions in the array. The `$and` (page 791) 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` (page 791) 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 } } ] }, { $se
```

```
db.inventory.update( { price: { $ne: 1.99, $exists: true } } , { $set: { qty: 15 } } )
```

Both update() (page 974) 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() (page 951), update() (page 974), $ne (page 789), $exists (page 794), $set (page 814).

$not

$not

Syntax: { field: { $not: { <operator-expression> } } }

$not (page 792) 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 (page 788) operator. { $lte: 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 (page 792) operator only affects *other operators* and cannot check fields and documents independently. So, use the $not (page 792) operator for logical disjunctions and the $ne (page 789) operator to test the contents of fields directly.

Consider the following behaviors when using the $not (page 792) operator:

- The operation of the $not (page 792) operator is consistent with the behavior of other operators but may yield unexpected results with some data types like arrays.
- The $not (page 792) operator does **not** support operations with the $regex (page 798) 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 you are using Python, you can write the above query with the PyMongo driver and Python's python:re.compile() method to compile a regular expression, as follows:

  ```
  import re
  for noMatch in db.inventory.find( { "item": { "$not": re.compile("^p.*") } } ):
      print noMatch
  ```

See also:

`find()` (page 951), `update()` (page 974), `$set` (page 814), `$gt` (page 786), `$regex` (page 798), PyMongo, *driver*.

$nor

$nor

Syntax: `{ $nor: [ { <expression1> }, { <expression2> }, ... { <expressionN> } ] }`

`$nor` (page 793) 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 `$nor` (page 793) expression is when the `$nor` (page 793) operator is used with the `$exists` (page 794) operator.

Consider the following query which uses only the `$nor` (page 793) 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 `$nor` (page 793) operator with the `$exists` (page 794) 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()` (page 951), `update()` (page 974), `$set` (page 814), `$exists` (page 794).

Element

Element Query Operators

Name	Description
`$exists` (page 794)	Matches documents that have the specified field.
`$mod` (page 794)	Performs a modulo operation on the value of a field and selects documents with a specified result.
`$type` (page 795)	Selects documents if a field is of the specified type.

$exists

`$exists`

Syntax: `{ field: { $exists: <boolean> } }`

`$exists` (page 794) selects the documents that contain the field if `<boolean>` is `true`. If `<boolean>` is `false`, the query only returns the documents that do not contain the field. Documents that contain the field but has the value `null` are *not* returned.

MongoDB *$exists* does **not** correspond to SQL operator `exists`. For SQL `exists`, refer to the `$in` (page 787) 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()` (page 951)
- `$nin` (page 789)
- `$and` (page 791)
- `$in` (page 787)
- *How do I query for fields that have null values?* (page 742)

$mod

`$mod`

Syntax: `{ field: { $mod: [ divisor, remainder ]} }`

`$mod` (page 794) 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 `0`, such as documents with `qty` value equal to `0` or `12`.

In some cases, you can query using the `$mod` (page 794) operator rather than the more expensive `$where` (page 797) operator. Consider the following example using the `$mod` (page 794) operator:

```
db.inventory.find( { qty: { $mod: [ 4, 0 ] } } )
```

The above query is less expensive than the following query which uses the `$where` (page 797) operator:

```
db.inventory.find( { $where: "this.qty % 4 == 0" } )
```

See also:

`find()` (page 951), `update()` (page 974), `$set` (page 814).

$type

`$type`

Syntax: `{ field: { $type: <BSON type> } }`

`$type` (page 795) 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` (page 795) 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` (page 797) 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
Undefined (deprecated)	6
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` (page 795) 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 (EST)"
{ "_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 1061):

```
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() (page 951), insert() (page 961), $where (page 797), *BSON*, *shard key*, *sharded cluster* .

JavaScript

JavaScript Query Operators

Name	Description
$where (page 797)	Matches documents that satisfy a JavaScript expression.
$regex (page 798)	Selects documents where values match a specified regular expression.

$where

$where

Use the $where (page 797) operator to pass either a string containing a JavaScript expression or a full JavaScript function to the query system. The $where (page 797) provides greater flexibility, but requires that the database processes the JavaScript expression or function for *each* document in the collection. Reference the document in the JavaScript expression or function using either `this` or `obj` .

> **Warning:**
> - Do not write to the database within the $where (page 797) JavaScript function.
> - $where (page 797) 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 (page 786), $in (page 787)).
> - In general, you should use $where (page 797) only when you can't express your query using another operator. If you must use $where (page 797), try to include at least one other standard query operator to filter the result set. Using $where (page 797) alone requires a table scan.

Consider the following examples:

```
db.myCollection.find( { $where: "this.credits == this.debits" } );
db.myCollection.find( { $where: "obj.credits == obj.debits" } );

db.myCollection.find( { $where: function() { return (this.credits == this.debits) } } );
db.myCollection.find( { $where: function() { return obj.credits == obj.debits; } } );
```

Additionally, if the query consists only of the $where (page 797) operator, you can pass in just the JavaScript expression or JavaScript functions, as in the following examples:

```
db.myCollection.find( "this.credits == this.debits || this.credits > this.debits" );

db.myCollection.find( function() { return (this.credits == this.debits || this.credits > this.de
```

You can include both the standard MongoDB operators and the $where (page 797) operator in your query, as in the following examples:

```
db.myCollection.find( { active: true, $where: "this.credits - this.debits < 0" } );
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < 0;
```

Using normal non-$where (page 797) query statements provides the following performance advantages:

- MongoDB will evaluate non-$where (page 797) components of query before $where (page 797) statements. If the non-$where (page 797) statements match no documents, MongoDB will not perform any query evaluation using $where (page 797).
- The non-$where (page 797) query statements may use an *index*.

Note: Changed in version 2.4.

In MongoDB 2.4, map-reduce operations (page 840), the group (page 836) command, and $where (page 797) operator expressions **cannot** access certain global functions or properties, such as db, that are available in the mongo (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your map-reduce operations (page 840), group (page 836) commands, or $where (page 797) operator expressions include any global shell functions or properties that are no longer available, such as db.

The following JavaScript functions and properties **are available** to map-reduce operations (page 840), the group (page 836) command, and $where (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
args MaxKey MinKey	assert() BinData() DBPointer() DBRef() doassert() emit() gc() HexData() hex_md5() isNumber() isObject() ISODate() isString()	Map() MD5() NumberInt() NumberLong() ObjectId() print() printjson() printjsononeline() sleep() Timestamp() tojson() tojsononeline() tojsonObject() UUID() version()

$regex

$regex

The $regex (page 798) operator provides regular expression capabilities for pattern matching *strings* in queries. MongoDB uses Perl compatible regular expressions (i.e. “PCRE.”)

You can specify regular expressions using regular expression objects or using the $regex (page 798) operator. 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 (page 798) uses “Perl Compatible Regular Expressions” (PCRE) as the matching engine.

$options

$regex (page 798) 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 (page 798) 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.

$regex (page 798) only provides the i and m options for the native JavaScript regular expression objects (e.g. http://docs.mongodb.org/manualacme.*corp/i). To use x and s you must use the "$regex (page 798)" operator with the "$options (page 798)" syntax.

To combine a regular expression match with other operators, you need to use the "$regex (page 798)" 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.

$regex (page 798) can only use an *index* efficiently when the regular expression has an anchor for the beginning (i.e. ^) of a string and is a case-sensitive match. Additionally, while http://docs.mongodb.org/manual^a/, http://docs.mongodb.org/manual^a.*/, and http://docs.mongodb.org/manual^a.*$/ match equivalent strings, 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.

Geospatial

Geospatial Query Operators

Operators

	Name	Description
Query Selectors	$geoWithin (page 800)	Selects geometries within a bounding *GeoJSON* geometry.
	$geoIntersects (page 801)	Selects geometries that intersect with a *GeoJSON* geometry.
	$near (page 801)	Returns geospatial objects in proximity to a point.
	$nearSphere (page 802)	Returns geospatial objects in proximity to a point on a sphere.

$geoWithin

$geoWithin

New in version 2.4: $geoWithin (page 800) replaces $within (page 801) which is deprecated.

The $geoWithin (page 800) operator is a geospatial query operator that queries for a defined point, line or shape that exists entirely within another defined shape. When determining inclusion, MongoDB considers the border of a shape to be part of the shape, subject to the precision of floating point numbers.

The $geoWithin (page 800) operator queries for inclusion in a *GeoJSON* polygon or a shape defined by legacy coordinate pairs.

The $geoWithin (page 800) operator does not return sorted results. As a result MongoDB can return $geoWithin (page 800) queries more quickly than geospatial $near (page 801) or $nearSphere (page 802) queries, which sort results.

The `2dsphere` and `2d` indexes both support the $geoWithin (page 800) operator.

Changed in version 2.2.3: $geoWithin (page 800) does not require a geospatial index. However, a geospatial index will improve query performance.

If querying for geometries that exist within a GeoJSON *polygon* on a sphere, pass the polygon to $geoWithin (page 800) using the $geometry (page 804) operator.

For a polygon with only an exterior ring use following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                            { $geometry :
                               { type : "Polygon" ,
                                 coordinates : [ [ [ <lng1>, <lat1> ] , [ <lng2>, <lat2> ] ... ]
                      } } } } )
```

Important: Specify coordinates in `longitude, latitude` order.

For a polygon with an exterior and interior ring use following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                            { $geometry :
                               { type : "Polygon" ,
                                 coordinates : [ [ [ <lng1>, <lat1> ] , [ <lng2>, <lat2> ] ... ]
                                                 [ [ <lngA>, <latA> ] , [ <lngB>, <latB> ] ... ]
                      } } } } )
```

The following example selects all indexed points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates: [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ]
                } } } } )
```

If querying for inclusion in a shape defined by legacy coordinate pairs on a plane, use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                            { <shape operator> : <coordinates>
                      } } } )
```

For the syntax of shape operators, see: `$box` (page 805), `$polygon` (page 806), `$center` (page 804) (defines a circle), and `$centerSphere` (page 805) (defines a circle on a sphere).

Note: Any geometry specified with *GeoJSON* to `$geoWithin` (page 800) queries, **must** fit within a single hemisphere. MongoDB interprets geometries larger than half of the sphere as queries for the smaller of the complementary geometries.

`$within`

Deprecated since version 2.4: `$geoWithin` (page 800) replaces `$within` (page 801) in MongoDB 2.4.

$geoIntersects

`$geoIntersects`

New in version 2.4.

The `$geoIntersects` (page 801) operator is a geospatial query operator that selects all locations that intersect with a *GeoJSON* object. A location intersects a GeoJSON object if the intersection is non-empty. This includes documents that have a shared edge. The `$geoIntersects` (page 801) operator uses spherical geometry.

The `2dsphere` geospatial index supports `$geoIntersects` (page 801).

To query for intersection, pass the GeoJSON object to `$geoIntersects` (page 801) through the `$geometry` (page 804) operator. Use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoIntersects :
                            { $geometry :
                               { type : "<GeoJSON object type>" ,
                                 coordinates : [ <coordinates> ]
                      } } } } )
```

Important: Specify coordinates in this order: **"longitude, latitude."**

The following example uses `$geoIntersects` (page 801) to select all indexed points and shapes that intersect with the polygon defined by the `coordinates` array.

```
db.places.find( { loc :
                  { $geoIntersects :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates: [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ] }
                } } } } )
```

Note: Any geometry specified with *GeoJSON* to `$geoIntersects` (page 801) queries, **must** fit within a single hemisphere. MongoDB interprets geometries larger than half of the sphere as queries for the smaller of the complementary geometries.

$near

`$near`

Changed in version 2.4.

The `$near` (page 801) operator specifies a point for which a *geospatial* query returns the 100 closest documents. The query sorts the documents from nearest to farthest.

The `$near` (page 801) operator can query for a *GeoJSON* point or for a point defined by legacy coordinate pairs.

The optional `$maxDistance` (page 804) operator limits a `$near` (page 801) query to return only those documents that fall within a maximum distance of a point. If you query for a GeoJSON point, specify `$maxDistance` (page 804) in meters. If you query for legacy coordinate pairs, specify `$maxDistance` (page 804) in radians.

The `$near` (page 801) operator requires a geospatial index. For GeoJSON points, use a `2dsphere` index. For legacy coordinate pairs, use a `2d` index.

For queries on GeoJSON data, use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $near :
                           { $geometry :
                              { type : "Point" ,
                                coordinates : [ <longitude> , <latitude> ] } },
                             $maxDistance : <distance in meters>
                      } } )
```

Important: Specify coordinates in this order: **"longitude, latitude."**

The following example selects the 100 documents with coordinates nearest to `[ 40 , 5 ]` and limits the maximum distance to 100 meters from the specified GeoJSON point:

```
db.places.find( { loc : { $near :
                           { $geometry :
                              { type : "Point" ,
                                coordinates: [ 40 , 5 ] } },
                             $maxDistance : 100
                } } )
```

For queries on legacy coordinate pairs, use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $near : [ <x> , <y> ] ,
                           $maxDistance: <distance>
                    } } )
```

Important: If you use longitude and latitude, specify **longitude first**.

The following example selects the 100 documents with coordinates nearest to `[ 40 , 5 ]`:

```
db.places.find( { loc :
                   { $near : [ 40 , 5 ] ,
                     $maxDistance : 10
                } } )
```

Note: You can further limit the number of results using `cursor.limit()` (page 985).

Specifying a batch size (i.e. `batchSize()` (page 978)) in conjunction with queries that use the `$near` (page 801) is not defined. See SERVER-5236 for more information.

$nearSphere

`$nearSphere`

New in version 1.8.

The `$nearSphere` (page 802) operator specifies a point for which a *geospatial* query returns the 100 closest documents, sorted from nearest to farthest. MongoDB calculates distances for `$nearSphere` (page 802) using spherical geometry.

The `$nearSphere` (page 802) operator queries for points defined by either *GeoJSON* objects or legacy coordinate pairs.

The optional `$maxDistance` (page 804) operator limits a `$nearSphere` (page 802) query to return only those documents that fall within a maximum distance of a point. If you use `$maxDistance` (page 804) on GeoJSON points, the distance is measured in meters. If you use `$maxDistance` (page 804) on legacy coordinate pairs, the distance is measured in radians.

The `$nearSphere` (page 802) operator requires a geospatial index. The `2dsphere` and `2d` indexes both support `$nearSphere` (page 802) with both legacy coordinate pairs and GeoJSON points..

Important: If you use longitude and latitude, specify **longitude first**.

For queries on GeoJSON data, use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $nearSphere :
                           { $geometry :
                              { type : "Point" ,
                                coordinates : [ <longitude> , <latitude> ] } ,
                             $maxDistance : <distance in meters>
                   } } } )
```

For queries on legacy coordinate pairs, use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $nearSphere: [ <x> , <y> ] ,
                           $maxDistance: <distance in radians>
                      } } )
```

The following example selects the 100 documents with legacy coordinates pairs nearest to `[ 40 , 5 ]`, as calculated by spherical geometry:

```
db.places.find( { loc :
                   { $nearSphere : [ 40 , 5 ]
                     $maxDistance : 10
                } } )
```

Geometry Specifiers

Name	Description
`$geometry` (page 804)	Specifies a geometry in *GeoJSON* format to geospatial query operators.
`$maxDistance` (page 804)	Specifies a distance to limit the results of `$near` (page 801) and `$nearSphere` (page 8 queries.
`$center` (page 804)	Specifies a circle using legacy coordinate pairs to `$geoWithin` (page 800) queries wher using planar geometry.
`$centerSphere` (page 805)	Specifies a circle using either legacy coordinate pairs or *GeoJSON* format for `$geoWith` (page 800) queries when using spherical geometry.
`$box` (page 805)	Specifies a rectangular box using legacy coordinate pairs for `$geoWithin` (page 800) queries.
`$polygon` (page 806)	Specifies a polygon to using legacy coordinate pairs for `$geoWithin` (page 800) querie
`$uniqueDocs` (page 807)	Modifies a `$geoWithin` (page 800) and `$near` (page 801) queries to ensure that even i document matches the query multiple times, the query returns the document once.

$geometry

$geometry

New in version 2.4.

The $geometry (page 804) operator specifies a *GeoJSON* for a geospatial query operators. For details on using $geometry (page 804) with an operator, see the operator:

- $geoWithin (page 800)
- $geoIntersects (page 801)
- $near (page 801)

$maxDistance

$maxDistance

The $maxDistance (page 804) operator constrains the results of a geospatial $near (page 801) or $nearSphere (page 802) query to the specified distance. The measuring units for the maximum distance are determined by the coordinate system in use. For *GeoJSON* point object, specify the distance in meters, not radians.

The 2d and 2dsphere geospatial indexes both support $maxDistance (page 804).

The following example query returns documents with location values that are 10 or fewer units from the point [100 , 100].

```
db.places.find( { loc : { $near : [ 100 , 100 ] ,
                          $maxDistance: 10 }
                } )
```

MongoDB orders the results by their distance from [100 , 100]. The operation returns the first 100 results, unless you modify the query with the cursor.limit() (page 985) method.

$center

$center

New in version 1.4.

The $center (page 804) operator specifies a circle for a *geospatial* $geoWithin (page 800) query. The query returns legacy coordinate pairs that are within the bounds of the circle. The operator does *not* return GeoJSON objects.

The query calculates distances using flat (planar) geometry.

The 2d geospatial index supports the $center (page 804) operator.

To use the $center (page 804) operator, specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius, as measured in the units used by the coordinate system

---

Important: If you use longitude and latitude, specify **longitude first**.

Use the following syntax:

```
{ <location field> : { $geoWithin : { $center : [ [ <x>, <y> ] , <radius> ] } } }
```

The following example query returns all documents that have coordinates that exist within the circle centered on [-74 , 40.74] and with a radius of 10:

```
db.places.find( { loc: { $geoWithin :
                          { $center : [ [-74, 40.74], 10 ] }
                } } )
```

Changed in version 2.2.3: Applications can use `$center` (page 804) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

$centerSphere

`$centerSphere`

New in version 1.8.

The `$centerSphere` (page 805) operator defines a circle for a *geospatial* query that uses spherical geometry. The query returns documents that are within the bounds of the circle.

You can use the `$centerSphere` (page 805) operator on both *GeoJSON* objects and legacy coordinate pairs.

The `2d` and `2dsphere` geospatial indexes both support `$centerSphere` (page 805).

To use `$centerSphere` (page 805), specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distances in a 2d Index Using Spherical Geometry* (page 349).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                         { $geoWithin :
                            { $centerSphere : [ [ <x>, <y> ] , <radius> ] }
                      } } )
```

Important: If you use longitude and latitude, specify **longitude first**.

The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude `88 W` and latitude `30 N`. The query converts the distance to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( { loc : { $geoWithin :
                           { $centerSphere :
                             [ [ 88 , 30 ] , 10 / 3959 ]
                } } } )
```

Changed in version 2.2.3: Applications can use `$centerSphere` (page 805) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

$box

`$box`

New in version 1.4.

The `$box` (page 805) operator specifies a rectangle for a *geospatial* `$geoWithin` (page 800) query. The query returns documents that are within the bounds of the rectangle, according to their point-based location data. The `$box` (page 805) operator returns documents based on *grid coordinates* (page 341) and does *not* query for GeoJSON shapes.

The query calculates distances using flat (planar) geometry. The `2d` geospatial index supports the `$box` (page 805) operator.

To use the `$box` (page 805) operator, you must specify the bottom left and top right corners of the rectangle in an array object. Use the following syntax:

```
{ <location field> : { $geoWithin : { $box :
                                       [ [ <bottom left coordinates> ] ,
                                         [ <upper right coordinates> ] ] } } }
```

Important: If you use longitude and latitude, specify **longitude first**.

The following example query returns all documents that are within the box having points at: `[ 0 , 0 ]`, `[ 0 , 100 ]`, `[ 100 , 0 ]`, and `[ 100 , 100 ]`.

```
db.places.find( { loc : { $geoWithin : { $box :
                                          [ [ 0 , 0 ] ,
                                            [ 100 , 100 ] ] } } } )
```

Changed in version 2.2.3: Applications can use `$box` (page 805) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

$polygon

`$polygon`

New in version 1.9.

The `$polygon` (page 806) operator specifies a polygon for a *geospatial* `$geoWithin` (page 800) query on legacy coordinate pairs. The query returns pairs that are within the bounds of the polygon. The operator does *not* query for GeoJSON objects.

The `$polygon` (page 806) operator calculates distances using flat (planar) geometry.

The `2d` geospatial index supports the `$polygon` (page 806) operator.

To define the polygon, specify an array of coordinate points. Use the following syntax:

```
{ <location field> : { $geoWithin : { $polygon : [ [ <x1> , <y1> ] ,
                                                   [ <x2> , <y2> ] ,
                                                   [ <x3> , <y3> ] ] } } }
```

Important: If you use longitude and latitude, specify **longitude first**.

The last point specified is always implicitly connected to the first. You can specify as many points, and therefore sides, as you like.

The following query returns all documents that have coordinates that exist within the polygon defined by `[ 0 , 0 ]`, `[ 3 , 6 ]`, and `[ 6 , 0 ]`:

```
db.places.find( { loc : { $geoWithin : { $polygon : [ [ 0 , 0 ] ,
                                                      [ 3 , 6 ] ,
                                                      [ 6 , 0 ] ] } } } )
```

Changed in version 2.2.3: Applications can use `$polygon` (page 806) *without* having a geospatial index. However, geospatial indexes support much faster queries than the unindexed equivalents. Before 2.2.3, a geospatial index *must* exist on a field holding coordinates before using any of the geospatial query operators.

$uniqueDocs

`$uniqueDocs`

New in version 2.0.

The `$uniqueDocs` (page 807) operator returns a document only once for a *geospatial* query if the document matches the query multiple times. A document might match a query multiple times if the documents contains multiple coordinate values.

You can use `$uniqueDocs` (page 807) only with the `$geoWithin` (page 800) and `$near` (page 801) operators. The `2d` geospatial index supports `$uniqueDocs` (page 807).

Example

Given a collection of addresses with documents in the following form:

```
{ addrs : [ { name : "H" , loc : [ 55.5 , 42.3 ] } , { name : "W" , loc : [ 32.3 , 44.2 ] } ] }
```

The following query would return the same document multiple times:

```
db.list.find( { "addrs.loc" : { $geoWithin : { $box : [ [ 0 , 0 ] , [ 100 , 100 ] ] } } } )
```

The following query would return each matching document only once:

```
db.list.find( { "addrs.loc" : { $geoWithin : { $box : [ [ 0 , 0 ] , [ 100 , 100 ] ], $uniqueDocs
```

Note: If you specify a value of `false` for `$uniqueDocs` (page 807), MongoDB will return multiple instances of a single document.

Geospatial Query Compatibility While numerous combinations of query operators are possible, the following table shows the recommended operators for different types of queries. The table uses the `$geoWithin` (page 800), `$geoIntersects` (page 801) and `$near` (page 801) operators.

Query Document	Geometry of the Query Condition	Surface Type for Query Calculation	Units for Query Calculation	Supported by this Index
Returns points, lines and polygons				
{ $geoWithin : { $geometry : <GeoJSON Polygon> } }	polygon	sphere	meters	2dsphere
{ $geoIntersects : { $geometry : <GeoJSON> } }	point, line or polygon	sphere	meters	2dsphere
{ $near : { $geometry : <GeoJSON Point>, $maxDistance : d } }	point	sphere	meters	2dsphere The index is required.
Returns points only				
{ $geoWithin : { $box : [[x1, y1], [x2, y2]] } }	rectangle	flat	flat units	2d
{ $geoWithin : { $polygon : [[x1, y1], [x1, y2], [x2, y2], [x2, y1]] } }	polygon	flat	flat units	2d
{ $geoWithin : { $center : [[x1, y1], r], } }	circular region	flat	flat units	2d
{ $geoWithin : { $centerSphere : [[x, y], radius] } }	circular region	sphere	radians	2d 2dsphere
{ $near : [x1, y1], $maxDistance : d }	point	flat / flat units	flat units	2d The index is required.

Array

Query Operator Array

Name	Description
`$elemMatch` (page 809)	Selects documents if element in the array field matches all the specified `$elemMatch` (page 809) condition.
`$size` (page 809)	Selects documents if the array field is a specified size.

$elemMatch (query) **See also:**

$elemMatch (projection) (page 824)

`$elemMatch`

New in version 1.4.

The `$elemMatch` (page 809) 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 809) 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 } ] }
```

$size

`$size`

The `$size` (page 809) 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 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` (page 809) with a value of 1, as follows:

```
db.collection.find( { field: { $size: 1 } } );
```

`$size` (page 809) 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` (page 809) portion of a query, although the other portions of a query can use indexes if applicable.

55.1.2 Update Operators

Fields

Field Update Operators

Name	Description
`$inc` (page 810)	Increments the value of the field by the specified amount.
`$rename` (page 810)	Renames a field.
`$setOnInsert` (page 813)	Sets the value of a field upon documentation creation during an upsert. Has no effect on update operations that modify existing documents.
`$set` (page 814)	Sets the value of a field in an existing document.
`$unset` (page 814)	Removes the specified field from an existing document.

$inc

`$inc`

The `$inc` (page 810) operator increments a value of a field by a specified amount. If the field does not exist, `$inc` (page 810) sets the field to the specified amount. `$inc` (page 810) accepts positive and negative incremental amounts.

The following example increments the value of `field1` by the value of `amount` for the *first* matching document in the collection where `field` equals `value`:

```
db.collection.update( { field: value },
                      { $inc: { field1: amount } } );
```

To update all matching documents in the collection, specify `multi:true` in the `update()` (page 974) method:

```
db.collection.update( { age: 20 }, { $inc: { age: 1 } }, { multi: true } );
db.collection.update( { name: "John" }, { $inc: { age: 2 } }, { multi: true } );
```

The first `update()` (page 974) operation increments the value of the `age` field by `1` for all documents in the collection that have an `age` field equal to `20`. The second operation increments the value of the `age` field by `2` for all documents in the collection with the `name` field equal to `"John"`.

$rename

`$rename`

New in version 1.7.2.

Syntax: `{$rename: { <old name1>: <new name1>, <old name2>: <new name2>, ... } }`

The `$rename` (page 810) 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` (page 810) operator removes that field and renames the field with the *old* field name to the *new* field name.

The $rename (page 810) 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 (page 810) 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 (page 810) operator using the *dot notation* (page 62) 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 (page 810) operator using the *dot notation* (page 62) 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 (page 810) 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 (page 810) 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 (page 810) 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 (page 810) 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 (page 810) 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 (page 810) operator does nothing.
  - **if** fields already exist with the new field names, the $rename (page 810) 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" }
}
```

$setOnInsert

$setOnInsert

New in version 2.4.

The $setOnInsert (page 813) operator assigns values to fields during an upsert (page 974) **only** when using the *upsert* option to the update() (page 974) operation performs an insert.

```
db.collection.update( <query>,
                      { $setOnInsert: { <field1>: <value1>, ... } },
                      { upsert: true }
                    )
```

Example

A collection named products contains no documents.

Then, the following upsert (page 974) operation performs an insert and applies the $setOnInsert (page 813) to set the field defaultQty to 100:

```
db.products.update(
                    { _id: 1 },
                    { $setOnInsert: { defaultQty: 100 } },
                    { upsert: true }
                  )
```

The products collection contains the newly-inserted document:

```
{ "_id" : 1, "defaultQty" : 100 }
```

Note: The $setOnInsert (page 813) operator only affects update() (page 974) operations with the *upsert* flag that perform an *insert* (page 75).

If the update() (page 974) has the upsert flag and performs an *update* (page 93), $setOnInsert (page 813) has no effect.

Example

A collection named products has the following document:

```
{ "_id" : 1, "defaultQty" : 100 }
```

The following update() (page 974) with the *upsert* flag operation performs an update:

```
db.products.update(
                    { _id: 1 },
                    { $setOnInsert: { defaultQty: 500, inStock: true },
                      $set: { item: "apple" } },
                    { upsert: true }
                  )
```

Because the update() (page 974) with *upsert* operation only performs an update, MongoDB ignores the $setOnInsert (page 813) operation and only applies the $set (page 814) operation.

The products collection now contains the following modified document:

```
{ "_id" : 1, "defaultQty" : 100, "item" : "apple" }
```

$set

$set

Use the $set (page 814) operator to set a particular value. The $set (page 814) 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.

$unset

$unset

The $unset (page 814) 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 the field in the $unset (page 814) 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 (page 814) operation (e.g. field1), then the statement has no effect on the document.

Array

Array Update Operators

Update Operators

Name	Description
$ (page 814)	Acts as a placeholder to update the first element that matches the query condition in an update.
$addToSet (page 815)	Adds elements to an existing array only if they do not already exist in the set.
$pop (page 816)	Removes the first or last item of an array.
$pullAll (page 816)	Removes multiple values from an array.
$pull (page 817)	Removes items from an array that match a query statement.
$pushAll (page 817)	*Deprecated.* Adds several items to an array.
$push (page 818)	Adds an item to an array.

$ (query)

$

Syntax: { "<array>.$" : value }

The positional `$` (page 814) operator identifies an element in an `array` field to update without explicitly specifying the position of the element in the array. To project, or return, an array element from a read operation, see the `$` (page 822) projection operator.

When used with the `update()` (page 974) method,

- the positional `$` (page 814) operator acts as a placeholder for the **first** element that matches the *query document* (page 64), and
- the `array` field **must** appear as part of the `query document`.

```
db.collection.update( { <array>: value ... }, { <update operator>: { "<array>.$" : value } } )
```

Consider a 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 `$` (page 814) 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 `$` (page 814) operator acts as a placeholder for the **first match** of the update *query document* (page 64).

The positional `$` (page 814) operator facilitates updates to arrays that contain embedded documents. Use the positional `$` (page 814) operator to access the fields in the embedded documents with the *dot notation* (page 62) on the `$` (page 814) 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 `$` (page 814) 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 } } )
```

Note:

- Do not use the positional operator `$` (page 814) with *upsert* operations because inserts will use the `$` as a field name in the inserted document.
- When used with the `$unset` (page 814) operator, the positional `$` (page 814) operator does not remove the matching element from the array but rather sets it to `null`.

See also:

`update()` (page 974), `$set` (page 814) and `$unset` (page 814)

$addToSet

`$addToSet`

The `$addToSet` (page 815) operator adds a value to an array only *if* the value is *not* in the array already. If the value *is* in the array, `$addToSet` (page 815) returns without modifying the array. Consider the following example:

```
db.collection.update( { field: value }, { $addToSet: { field: value1 } } );
```

Here, `$addToSet` (page 815) appends `value1` to the array stored in `field`, *only if* `value1` is not already a member of this array.

Note: `$addToSet` (page 815) only ensures that there are no duplicate items *added* to the set and does not affect existing duplicate elements. `$addToSet` (page 815) does not guarantee a particular ordering of elements in the modified set.

Use the `$each` (page 819) modifier with the `$addToSet` (page 815) operator to add multiple values to an array <field> if the values do not exist in the <field>.

```
db.collection.update( <query>,
                      {
                        $addToSet: { <field>: { $each: [ <value1>, <value2> ... ] } }
                      }
                    )
```

See also:

`$push` (page 818)

$pop

`$pop`

The `$pop` (page 816) operator removes the first or last element of an array. Pass `$pop` (page 816) 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` (page 816) behaviors:

- The `$pop` (page 816) operation fails if `field` is not an array.
- `$pop` (page 816) will successfully remove the last item in an array. `field` will then hold an empty array.

New in version 1.1.

$pullAll

`$pullAll`

The `$pullAll` (page 816) operator removes multiple values from an existing array. `$pullAll` (page 816) provides the inverse operation of the `$pushAll` (page 817) operator. Consider the following example:

```
db.collection.update( { field: value }, { $pullAll: { field1: [ value1, value2, value3 ] } } );
```

Here, `$pullAll` (page 816) removes `[ value1, value2, value3 ]` from the array in `field1`, in the document that matches the query statement `{ field: value }` in `collection`.

$pull

`$pull`

The `$pull` (page 817) operator removes all instances of a value from an existing array, as in the following prototype:

```
db.collection.update( { field: <query> }, { $pull: { field: <query> } } );
```

`$pull` (page 817) removes items from the array in the field named `field` that match the query in the `$pull` (page 817) statement.

If a value (i.e. `<value>`) exists multiple times in an array, `$pull` (page 817) will remove all instances of the value.

Example

Given the following document in the `cpuinfo` collection:

```
{ flags: ['vme', 'de', 'pse', 'tsc', 'msr', 'pae', 'mce' ] }
```

The following operation will remove the `msr` value from the `flags` array:

```
db.cpuinfo.update( { flags: 'msr' }, { $pull: { flags: 'msr' } } )
```

Example

Given the following document in the `profiles` collection:

```
{ votes: [ 3, 5, 6, 7, 7, 8 ] }
```

The following operation will remove all occurrences of `7` from the `votes` array.

```
db.profiles.update( { votes: 3 }, { $pull: { votes: 7 } } )
```

Therefore, the `votes` array would resemble the following:

{ votes: [3, 5, 6, 8] }

Conversely, the following operation will remove all items from the array that are larger than `6`:

```
db.profiles.update( { votes: 3 }, { $pull: { votes: { $gt: 6 } } } )
```

Therefore, the `votes` array would resemble the following:

```
{ votes: [ 3, 5, 6 ] }
```

$pushAll

`$pushAll`

Deprecated since version 2.4: Use the `$push` (page 818) operator with `$each` (page 819) instead.

The `$pushAll` (page 817) operator is similar to the `$push` (page 818) 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` (page 817) 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, `$pushAll` (page 817) will behave as `$push` (page 818).

$push

`$push`

The `$push` (page 818) operator appends a specified value to an array.

```
db.collection.update( <query>,
                      { $push: { <field>: <value> } }
                    )
```

The following example appends `89` to the `scores` array for the first document where the `name` field equals `joe`:

```
db.students.update(
                    { name: "joe" },
                    { $push: { scores: 89 } }
                  )
```

Note:

- If the field is absent in the document to update, `$push` (page 818) adds the array field with the value as its element.
- If the field is **not** an array, the operation will fail.
- If the value is an array, `$push` (page 818) appends the whole array as a *single* element. To add each element of the value separately, use `$push` (page 818) with the `$each` (page 819) modifier.

  The following example appends each element of `[ 90, 92, 85 ]` to the `scores` array for the document where the `name` field equals `joe`:

  ```
  db.students.update(
                      { name: "joe" },
                      { $push: { scores: { $each: [ 90, 92, 85 ] } } }
                    )
  ```

  Changed in version 2.4: MongoDB adds support for the `$each` (page 819) modifier to the `$push` (page 818) operator. Before 2.4, use `$pushAll` (page 817) for similar functionality.

Changed in version 2.4: You can use the `$push` (page 818) operator with the following modifiers:

- `$each` (page 819) appends multiple values to the array field,
- `$slice` (page 819), which is only available with `$each` (page 819), limits the number of array elements, and
- `$sort` (page 820), which is only available with `$each` (page 819), orders elements of the array. `$sort` (page 820) can only order array elements that are documents.

The following example uses:

- the `$each` (page 819) modifier to append documents to the `quizzes` array,
- the `$sort` (page 820) modifier to sort all the elements of the modified `quizzes` array by the ascending `score` field, and
- the `$slice` (page 819) modifier to keep only the **last** five sorted elements of the `quizzes` array.

```
db.students.update( { name: "joe" },
                    { $push: { quizzes: { $each: [ { id: 3, score: 8 },
                                                   { id: 4, score: 7 },
                                                   { id: 5, score: 6 } ],
                                          $sort: { score: 1 },
                                          $slice: -5
```

```
                                }
                        }
                }
        )
```

Update Operator Modifiers

Name	Description
$each (page 819)	Modifies the $push (page 818) and $addToSet (page 815) operators to append items for array updates.
$slice (page 819)	Modifies the $push (page 818) operator to limit the size of updated arrays.
$sort (page 820)	Modifies the $push (page 818) operator to reorder documents stored in an array.

$each

Note: The $each (page 819) modifier is only used with the $addToSet (page 815) and $push (page 818) operators. See the documentation of $addToSet (page 815) and $push (page 818) for more information.

$each

The $each (page 819) modifier is available for use with the $addToSet (page 815) operator and the $push (page 818) operator.

Use the $each (page 819) modifier with the $addToSet (page 815) operator to add multiple values to an array `<field>` if the values do not exist in the `<field>`.

```
db.collection.update( <query>,
                      {
                        $addToSet: { <field>: { $each: [ <value1>, <value2> ... ] } }
                      }
                    )
```

Use the $each (page 819) modifier with the $push (page 818) operator to append multiple values to an array `<field>`.

```
db.collection.update( <query>,
                      {
                        $push: { <field>: { $each: [ <value1>, <value2> ... ] } }
                      }
                    )
```

Changed in version 2.4: MongoDB adds support for the $each (page 819) modifier to the $push (page 818) operator.

$slice

$slice

New in version 2.4.

The $slice (page 819) modifier limits the number of array elements during a $push (page 818) operation. To project, or return, a specified number of array elements from a read operation, see the $slice (page 826) projection operator instead.

To use the $slice (page 819) modifier, it must appear with the $each (page 819) modifier, *and* the $each (page 819) modifier must be the first modifier for the $push (page 818) operation.

```
db.collection.update( <query>,
                      { $push: {
                                 <field>: {
```

```
                                                  $each: [ <value1>, <value2>, ... ],
                                                  $slice: <num>
                                                }
                                    }
                           }
                         )
```

The <num> is either a **negative** number or **zero**:

- If <num> is **negative**, the array <field> contains only the last <num> elements.
- If <num> is **zero**, the array <field> is an empty array.

```
db.students.update( { _id: 2 },
                    { $push: { grades: {
                                             $each: [ 80, 78, 86 ],
                                             $slice: -5
                                           }
                               }
                    }
                  )
```

$sort

$sort

New in version 2.4.

The $sort (page 820) modifier orders the elements of an array during a $push (page 818) operation. The elements of the array **must** be documents.

$sort (page 820) modifies $push (page 818) updates that use the $each (page 819) and $slice (page 819) modifiers, where $each (page 819) is the first modifier for the $push (page 818) operation.

```
db.collection.update( <query>,
                      { $push: {
                                 <field>: {
                                            $each: [ <document1>,
                                                     <document2>,
                                                     ...
                                                   ],
                                            $slice: <num>,
                                            $sort: <sort document>,
                                          }
                               }
                      }
                    )
```

Important: The <sort document> only accesses the fields from the elements in the array and does **not** refer to the array <field>.

Consider the following example where the collection students contain the following document:

```
{ "_id": 3,
  "name": "joe",
  "quizzes": [
               { "id" : 1, "score" : 6 },
               { "id" : 2, "score" : 9 }
             ]
}
```

The following update appends additional documents to the `quizzes` array, sorts all the elements of the array by ascending `score` field, and slices the array to keep the last five elements:

```
db.students.update( { name: "joe" },
                    { $push: { quizzes: { $each: [ { id: 3, score: 8 },
                                                   { id: 4, score: 7 },
                                                   { id: 5, score: 6 } ],
                                          $sort: { score: 1 },
                                          $slice: -5
                                        }
                             }
                    }
                  )
```

After the update, the array elements are in order of ascending `score` field.:

```
{
  "_id" : 3,
  "name" : "joe",
  "quizzes" : [
                { "id" : 1, "score" : 6 },
                { "id" : 5, "score" : 6 },
                { "id" : 4, "score" : 7 },
                { "id" : 3, "score" : 8 },
                { "id" : 2, "score" : 9 }
              ]
}
```

Bitwise

Bitwise Update Operator

Name	Description
`$bit` (page 821)	Performs bitwise `AND` and `OR` updates of integer values.

$bit

$bit

The `$bit` (page 821) operator performs a bitwise update of a field. Only use this with integer fields, as in the following examples:

```
db.collection.update( { field: NumberInt(1) }, { $bit: { field: { and: NumberInt(5) } } } );

db.collection.update( { field: NumberInt(1) }, { $bit: { field: { or: NumberInt(5) } } } );
```

Here, the `$bit` (page 821) operator updates the integer value of the field named `field`: in the first example with a bitwise `and: 5` operation; and in the second example with a bitwise `or: 5` operation. `$bit` (page 821) only works with integers.

`$bit` (page 821) only supports `AND` and `OR` bitwise operations.

Note: All numbers in the `mongo` (page 1066) shell are doubles, not integers. Use the `NumberInt()` constructor to specify integers. See *NumberInt* (page 611) for more information.

Isolation

Isolation Update Operator

Name	Description
`$isolated` (page 822)	Modifies behavior of multi-updates to improve the isolation of the operation.

$isolated

`$isolated`

`$isolated` (page 822) isolation operator **isolates** a write operation that affects multiple documents from other write operations.

Note: The `$isolated` (page 822) isolation operator does **not** provide "all-or-nothing" atomicity for write operations.

Consider the following example:

```
db.foo.update( { field1 : 1 , $isolated : 1 }, { $inc : { field2 : 1 } } , { multi: true } )
```

Without the `$isolated` (page 822) operator, multi-updates will allow other operations to interleave with these updates. If these interleaved operations contain writes, the update operation may produce unexpected results. By specifying `$isolated` (page 822) you can guarantee isolation for the entire multi-update.

Warning: `$isolated` (page 822) does not work with *sharded clusters*.

See also:

See `db.collection.update()` (page 974) for more information about the `db.collection.update()` (page 974) method.

`$atomic`

Deprecated since version 2.2: The `$isolated` (page 822) replaces `$atomic` (page 822).

55.1.3 Projection Operators

Projection Operators

Name	Description
`$` (page 822)	Projects the first element in an array that matches the query condition.
`$elemMatch` (page 824)	Projects only the first element from an array that matches the specified `$elemMatch` (page 824) condition.
`$slice` (page 826)	Limits the number of elements projected from an array. Supports skip and limit slices.

$ (projection)

`$`

The positional `$` (page 822) operator limits the contents of the `<array>` field that is included in the query results to contain the **first** matching element. To specify an array element to update, see the *positional $ operator for updates* (page 814).

Used in the *projection* document of the `find()` (page 951) method or the `findOne()` (page 955) method:

- The `$` (page 822) projection operator limits the content of the `<array>` field to the **first** element that matches the *query document* (page 42).
- The `<array>` field **must** appear in the *query document* (page 42)

```
db.collection.find( { <array>: <value> ... },
                    { "<array>.$": 1 } )
db.collection.find( { <array.field>: <value> ...},
                    { "<array>.$": 1 } )
```

The `<value>` can be documents that contains *query operator expressions* (page 785).

- Only **one** positional `$` (page 822) operator can appear in the projection document.
- Only **one** array field can appear in the *query document* (page 42); i.e. the following query is **incorrect**:

```
db.collection.find( { <array>: <value>, <someOtherArray>: <value2> },
                    { "<array>.$": 1 } )
```

Example

A collection `students` contains the following documents:

```
{ "_id" : 1, "semester" : 1, "grades" : [ 70, 87, 90 ] }
{ "_id" : 2, "semester" : 1, "grades" : [ 90, 88, 92 ] }
{ "_id" : 3, "semester" : 1, "grades" : [ 85, 100, 90 ] }
{ "_id" : 4, "semester" : 2, "grades" : [ 79, 85, 80 ] }
{ "_id" : 5, "semester" : 2, "grades" : [ 88, 88, 92 ] }
{ "_id" : 6, "semester" : 2, "grades" : [ 95, 90, 96 ] }
```

In the following query, the projection `{ "grades.$": 1 }` returns only the first element greater than or equal to `85` for the `grades` field.

```
db.students.find( { semester: 1, grades: { $gte: 85 } },
                  { "grades.$": 1 } )
```

The operation returns the following documents:

```
{ "_id" : 1, "grades" : [ 87 ] }
{ "_id" : 2, "grades" : [ 90 ] }
{ "_id" : 3, "grades" : [ 85 ] }
```

Although the array field `grades` may contain multiple elements that are greater than or equal to `85`, the `$` (page 822) projection operator returns only the first matching element from the array.

Important: When the `find()` (page 951) method includes a `sort()` (page 991), the `find()` (page 951) method applies the `sort()` (page 991) to order the matching documents **before** it applies the positional `$` (page 822) projection operator.

If an array field contains multiple documents with the same field name and the `find()` (page 951) method includes a `sort()` (page 991) on that repeating field, the returned documents may not reflect the sort order because the sort was applied to the elements of the array before the `$` (page 822) projection operator.

Example

A `students` collection contains the following documents where the `grades` field is an array of documents; each document contain the three field names `grade`, `mean`, and `std`:

```
{ "_id" : 7, semester: 3, "grades" : [ { grade: 80, mean: 75, std: 8 },
                                       { grade: 85, mean: 90, std: 5 },
                                       { grade: 90, mean: 85, std: 3 } ] }

{ "_id" : 8, semester: 3, "grades" : [ { grade: 92, mean: 88, std: 8 },
                                       { grade: 78, mean: 90, std: 5 },
                                       { grade: 88, mean: 85, std: 3 } ] }
```

In the following query, the projection `{ "grades.$": 1 }` returns only the first element with the `mean` greater than `70` for the `grades` field. The query also includes a `sort()` (page 991) to order by ascending `grades.grade` field:

```
db.students.find( { "grades.mean": { $gt: 70 } },
                  { "grades.$": 1 }
                ).sort( { "grades.grade": 1 } )
```

The `find()` (page 951) method sorts the matching documents **before** it applies the `$` (page 822) projection operator on the `grades` array. Thus, the results with the projected array elements do not reflect the ascending `grades.grade` sort order:

```
{ "_id" : 8, "grades" : [ { "grade" : 92, "mean" : 88, "std" : 8 } ] }
{ "_id" : 7, "grades" : [ { "grade" : 80, "mean" : 75, "std" : 8 } ] }
```

Note: Since only **one** array field can appear in the query document, if the array contains documents, to specify criteria on multiple fields of these documents, use the *$elemMatch (query)* (page 809) operator, e.g.:

```
db.students.find( { grades: { $elemMatch: {
                                            mean: { $gt: 70 },
                                            grade: { $gt:90 }
                                          } } },
                  { "grades.$": 1 } )
```

See also:

`$elemMatch (projection)` (page 824)

$elemMatch (projection)

See also:

$elemMatch (query) (page 809)

$elemMatch

New in version 2.2.

The `$elemMatch` (page 824) projection operator limits the contents of an array field that is included in the query results to contain only the array element that matches the `$elemMatch` (page 824) condition.

Note:

- The elements of the array are documents.
- If multiple elements match the `$elemMatch` (page 824) condition, the operator returns the **first** matching element in the array.
- The `$elemMatch` (page 824) projection operator is similar to the positional `$` (page 822) projection operator.

The examples on the $elemMatch (page 824) projection operator assumes a collection school with the following documents:

```
{
 _id: 1,
 zipcode: 63109,
 students: [
              { name: "john", school: 102, age: 10 },
              { name: "jess", school: 102, age: 11 },
              { name: "jeff", school: 108, age: 15 }
           ]
}
{
 _id: 2,
 zipcode: 63110,
 students: [
              { name: "ajax", school: 100, age: 7 },
              { name: "achilles", school: 100, age: 8 },
           ]
}

{
 _id: 3,
 zipcode: 63109,
 students: [
              { name: "ajax", school: 100, age: 7 },
              { name: "achilles", school: 100, age: 8 },
           ]
}

{
 _id: 4,
 zipcode: 63109,
 students: [
              { name: "barney", school: 102, age: 7 },
           ]
}
```

Example

The following find() (page 951) operation queries for all documents where the value of the zipcode field is 63109. The $elemMatch (page 824) projection returns only the **first** matching element of the students array where the school field has a value of 102:

```
db.schools.find( { zipcode: 63109 },
                 { students: { $elemMatch: { school: 102 } } } )
```

The operation returns the following documents:

```
{ "_id" : 1, "students" : [ { "name" : "john", "school" : 102, "age" : 10 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "barney", "school" : 102, "age" : 7 } ] }
```

- For the document with _id equal to 1, the students array contains multiple elements with the school field equal to 102. However, the $elemMatch (page 824) projection returns only the first matching element from the array.

- The document with _id equal to 3 does not contain the students field in the result since no element in its students array matched the $elemMatch (page 824) condition.

The $elemMatch (page 824) projection can specify criteria on multiple fields:

Example

The following find() (page 951) operation queries for all documents where the value of the zipcode field is 63109. The projection includes the **first** matching element of the students array where the school field has a value of 102 **and** the age field is greater than 10:

```
db.schools.find( { zipcode: 63109 },
                 { students: { $elemMatch: { school: 102, age: { $gt: 10} } } } )
```

The operation returns the three documents that have zipcode equal to 63109:

```
{ "_id" : 1, "students" : [ { "name" : "jess", "school" : 102, "age" : 11 } ] }
{ "_id" : 3 }
{ "_id" : 4 }
```

Documents with _id equal to 3 and _id equal to 4 do not contain the students field since no element matched the $elemMatch (page 824) criteria.

When the find() (page 951) method includes a sort() (page 991), the find() (page 951) method applies the sort() (page 991) to order the matching documents **before** it applies the projection.

If an array field contains multiple documents with the same field name and the find() (page 951) method includes a sort() (page 991) on that repeating field, the returned documents may not reflect the sort order because the sort() (page 991) was applied to the elements of the array before the $elemMatch (page 824) projection.

Example

The following query includes a sort() (page 991) to order by descending students.age field:

```
db.schools.find(
                 { zipcode: 63109 },
                 { students: { $elemMatch: { school: 102 } } }
               ).sort( { "students.age": -1 } )
```

The operation applies the sort() (page 991) to order the documents that have the field zipcode equal to 63109 and then applies the projection. The operation returns the three documents in the following order:

```
{ "_id" : 1, "students" : [ { "name" : "john", "school" : 102, "age" : 10 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "barney", "school" : 102, "age" : 7 } ] }
```

See also:

$ (projection) (page 822) operator

$slice (projection)

$slice

The $slice (page 826) operator controls the number of items of an array that a query returns. For information on limiting the size of an array during an update with $push (page 818), see the $slice (page 819) modifier instead.

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 826) 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 826) 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` (page 826). 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.

55.1.4 Meta-Query Operators

Query Modification Operators

Introduction

In addition to the *MongoDB Query Operators* (page 785), there are a number of "meta" operators that you can modify the output or behavior of a query. On the server, MongoDB treats the query and the options as a single object. The `mongo` (page 1066) shell and driver interfaces may provide *cursor methods* (page 977) that wrap these options. When possible, use these methods; otherwise, you can add these options using either of the following syntax:

```
db.collection.find( { <query> } )._addSpecial( <option> )
db.collection.find( { $query: { <query> }, <option> } )
```

Operators

Modifiers Many of these operators have corresponding *methods in the shell* (page 977). These methods provide a straightforward and user-friendly interface and are the preferred way to add these options.

Name	Description
`$comment` (page 828)	Adds a comment to the query to identify queries in the *database profiler* output.
`$explain` (page 828)	Forces MongoDB to report on query execution plans. See `explain()` (page 979).
`$hint` (page 829)	Forces MongoDB to use a specific index. See `hint()` (page 985)
`$maxScan` (page 829)	Limits the number of documents a cursor will return for a query. See `limit()` (page 985).
`$max` (page 830)	Specifies a minimum exclusive upper limit for the index to use in a query. See `max()` (page 986).
`$min` (page 830)	Specifies a minimum inclusive lower limit for the index to use in a query. See `min()` (page 987).
`$orderby` (page 831)	Returns a cursor with documents sorted according to a sort specification. See `sort()` (page 991).
`$returnKey` (page 832)	Forces the cursor to only return fields included in the index.
`$showDiskLoc` (page 832)	Modifies the documents returned to include references to the on-disk location of each document.
`$snapshot` (page 832)	Forces the query to use the index on the `_id` field. See `snapshot()` (page 990).

$comment

`$comment`

The `$comment` (page 828) makes it possible to attach a comment to a query. Because these comments propagate to the `profile` (page 907) log, adding `$comment` (page 828) modifiers can make your profile data much easier to interpret and trace. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$comment", <comment> )
db.collection.find( { $query: { <query> }, $comment: <comment> } )
```

$explain

`$explain`

The `$explain` (page 828) operator provides information on the query plan. It returns a document that describes the process and indexes used to return the query. This may provide useful insight when attempting to optimize a query. For details on the output, see *cursor.explain()* (page 979).

You can specify the `$explain` (page 828) operator in either of the following forms:

```
db.collection.find()._addSpecial( "$explain", 1 )
db.collection.find( { $query: {}, $explain: 1 } )
```

You also can specify `$explain` (page 828) through the `explain()` (page 979) method in the `mongo` (page 1066) shell:

```
db.collection.find().explain()
```

`$explain` (page 828) runs the actual query to determine the result. Although there are some differences between running the query with `$explain` (page 828) and running without, generally, the performance will be similar between the two. So, if the query is slow, the `$explain` (page 828) operation is also slow.

Additionally, the `$explain` (page 828) operation reevaluates a set of candidate query plans, which may cause the `$explain` (page 828) operation to perform differently than a normal query. As a result, these operations generally provide an accurate account of *how* MongoDB would perform the query, but do not reflect the length of these queries.

To determine the performance of a particular index, you can use `hint()` (page 985) and in conjunction with `explain()` (page 979), as in the following example:

```
db.products.find().hint( { type: 1 } ).explain()
```

When you run `explain()` (page 979) with `hint()` (page 985), the query optimizer does not reevaluate the query plans.

Note: In some situations, the `explain()` (page 979) operation may differ from the actual query plan used by MongoDB in a normal query.

The `explain()` (page 979) operation evaluates the set of query plans and reports on the winning plan for the query. In normal operations the query optimizer caches winning query plans and uses them for similar related queries in the future. As a result MongoDB may sometimes select query plans from the cache that are different from the plan displayed using `explain()` (page 979).

See also:

- `cursor.explain()` (page 979)
- *Optimization Strategies for MongoDB* (page 576) page for information regarding optimization strategies.
- *Analyze Performance of Database Operations* (page 175) tutorial for information regarding the database profile.
- *Current Operation Reporting* (page 998)

$hint

`$hint`

The `$hint` (page 829) operator forces the *query optimizer* (page 48) to use a specific index to fulfill the query. Specify the index either by the index name or by the index specification document. See *Index Specification Documents* (page 66) for information on index specification documents.

Use `$hint` (page 829) for testing query performance and indexing strategies. The `mongo` (page 1066) shell provides a helper method `hint()` (page 985) for the `$hint` (page 829) operator.

Consider the following operation:

```
db.users.find().hint( { age: 1 } )
```

This operation returns all documents in the collection named `users` using the index on the `age` field.

You can also specify a hint using either of the following forms:

```
db.users.find()._addSpecial( "$hint", { age : 1 } )
db.users.find( { $query: {}, $hint: { age : 1 } } )
```

Note: To combine `$explain` (page 828) and `$hint` (page 829) operations, use the following form:

```
db.users.find( { $query: {}, $hint: { age : 1 } } )
```

You must add the `$explain` (page 828) option to the document, as in the following:

```
db.users.find( { $query: {}, $hint: { age : 1 }, $explain: 1 } )
```

$maxScan

`$maxScan`

Constrains the query to only scan the specified number of documents when fulfilling the query. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$maxScan" , <number> )
db.collection.find( { $query: { <query> }, $maxScan: <number> } )
```

Use this modifier to prevent potentially long running queries from disrupting performance by scanning through too much data.

$max

$max

Specify a `$max` (page 830) value to specify the *exclusive* upper bound for a specific index in order to constrain the results of `find()` (page 951). The `mongo` (page 1066) shell provides the `cursor.max()` (page 986) wrapper method:

```
db.collection.find( { <query> } ).max( { field1: <max value>, ... fieldN: <max valueN> } )
```

You can also specify the option with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$max", { field1: <max value1>, ... fieldN: <max
db.collection.find( { $query: { <query> }, $max: { field1: <max value1>, ... fieldN: <max valueN
```

The `$max` (page 830) specifies the upper bound for *all* keys of a specific index *in order*.

Consider the following operations on a collection named `collection` that has an index `{ age: 1 }`:

```
db.collection.find( { <query> } ).max( { age: 100 } )
```

This operation limits the query to those documents where the field `age` is less than `100` using the index `{ age: 1 }`.

You can explicitly specify the corresponding index with `cursor.hint()` (page 985). Otherwise, MongoDB selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

Consider a collection named `collection` that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using `cursor.hint()` (page 985), MongoDB may select either index for the following operation:

```
db.collection.find().max( { age: 50, type: 'B' } )
```

Use `$max` (page 830) alone or in conjunction with `$min` (page 830) to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

Note: Because `cursor.max()` (page 986) requires an index on a field, and forces the query to use this index, you may prefer the `$lt` (page 788) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).max( { age: 25 } )
```

The query uses the index on the `age` field, even if the index on `_id` may be better.

$min

$min

Specify a $min (page 830) value to specify the *inclusive* lower bound for a specific index in order to constrain the results of find() (page 951). The mongo (page 1066) shell provides the cursor.min() (page 987) wrapper method:

```
db.collection.find( { <query> } ).min( { field1: <min value>, ... fieldN: <min valueN>} )
```

You can also specify the option with either of the two forms:

```
db.collection.find( { <query> } )._addSpecial( "$min", { field1: <min value1>, ... fieldN: <min
db.collection.find( { $query: { <query> }, $min: { field1: <min value1>, ... fieldN: <min valueN
```

The $min (page 830) specifies the lower bound for *all* keys of a specific index *in order*.

Consider the following operations on a collection named `collection` that has an index `{ age: 1 }`:

```
db.collection.find().min( { age: 20 } )
```

These operations limit the query to those documents where the field `age` is at least `20` using the index `{ age: 1 }`.

You can explicitly specify the corresponding index with cursor.hint() (page 985). Otherwise, MongoDB selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

Consider a collection named `collection` that has the following two indexes:

```
{ age: 1, type: -1 }
{ age: 1, type: 1 }
```

Without explicitly using cursor.hint() (page 985), it is unclear which index the following operation will select:

```
db.collection.find().min( { age: 20, type: 'C' } )
```

You can use $min (page 830) in conjunction with $max (page 830) to limit results to a specific range for the *same* index, as in the following example:

```
db.collection.find().min( { age: 20 } ).max( { age: 25 } )
```

Note: Because cursor.min() (page 987) requires an index on a field, and forces the query to use this index, you may prefer the $gte (page 787) operator for the query if possible. Consider the following example:

```
db.collection.find( { _id: 7 } ).min( { age: 25 } )
```

The query will use the index on the `age` field, even if the index on `_id` may be better.

$orderby

$orderby

The $orderby (page 831) operator sorts the results of a query in ascending or descending order.

The mongo (page 1066) shell provides the cursor.sort() (page 991) method:

```
db.collection.find().sort( { age: -1 } )
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$orderby", { age : -1 } )
db.collection.find( { $query: {}, $orderby: { age : -1 } } )
```

These examples return all documents in the collection named `collection` sorted by the `age` field in descending order. Specify a value to `$orderby` (page 831) 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 an index for the specified key pattern, use `$orderby` (page 831) in conjunction with `$maxScan` (page 829) and/or `cursor.limit()` (page 985) to avoid requiring MongoDB to perform a large in-memory sort. The `cursor.limit()` (page 985) increases the speed and reduces the amount of memory required to return this query by way of an optimized algorithm.

$returnKey

`$returnKey`

Only return the index field or fields for the results of the query. If `$returnKey` (page 832) is set to `true` and the query does not use an index to perform the read operation, the returned documents will not contain any fields. Use one of the following forms:

```
db.collection.find( { <query> } )._addSpecial( "$returnKey", true )
db.collection.find( { $query: { <query> }, $returnKey: true } )
```

$showDiskLoc

`$showDiskLoc`

`$showDiskLoc` (page 832) option adds a field `$diskLoc` to the returned documents. The `$diskLoc` field contains the disk location information.

The `mongo` (page 1066) shell provides the `cursor.showDiskLoc()` (page 990) method:

```
db.collection.find().showDiskLoc()
```

You can also specify the option in either of the following forms:

```
db.collection.find( { <query> } )._addSpecial("$showDiskLoc" , true)
db.collection.find( { $query: { <query> }, $showDiskLoc: true } )
```

$snapshot

`$snapshot`

The `$snapshot` (page 832) operator prevents the cursor from returning a document more than once because an intervening write operation results in a move of the document.

Even in snapshot mode, objects inserted or deleted during the lifetime of the cursor may or may not be returned.

The `mongo` (page 1066) shell provides the `cursor.snapshot()` (page 990) method:

```
db.collection.find().snapshot()
```

You can also specify the option in either of the following forms:

```
db.collection.find()._addSpecial( "$snapshot", true )
db.collection.find( { $query: {}, $snapshot: true } )
```

The `$snapshot` (page 832) operator traverses the index on the `_id` field [1].

Warning:

- You cannot use `$snapshot` (page 832) with *sharded collections*.
- Do **not** use `$snapshot` (page 832) with `$hint` (page 829) or `$orderby` (page 831) (or the corresponding `cursor.hint()` (page 985) and `cursor.sort()` (page 991) methods.)

[1] You can achieve the `$snapshot` (page 832) isolation behavior using any *unique* index on invariable fields.

Sort Order

Name	Description
`$natural` (page 833)	A special sort order that orders documents using the order of documents on disk.

$natural

`$natural`

Use the `$natural` (page 833) operator to use *natural order* for the results of a sort operation. Natural order refers to the order of documents in the file on disk.

The `$natural` (page 833) operator uses the following syntax to return documents in the order they exist on disk:

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

Use `-1` to return documents in the reverse order as they occur on disk:

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

See also:

`cursor.sort()` (page 991)

55.2 Database Commands

All command documentation outlined below describes a command and its available parameters and provides a document template or prototype for each command. Some command documentation also includes the relevant `mongo` (page 1066) shell helpers.

55.2.1 User Commands

Aggregation Commands

Aggregation Commands

Command Index

Name	Description
`aggregate` (page 834)	Performs *aggregation tasks* (page 247) such as group using the aggregation framewor
`count` (page 834)	Counts the number of documents in a collection.
`distinct` (page 835)	Displays the distinct values found for a specified key in a collection.
`group` (page 836)	Groups documents in a collection by the specified key and performs simple aggregati
`mapReduce` (page 840)	Performs *map-reduce* (page 291) aggregation for large data sets.

aggregate

aggregate

New in version 2.1.0.

aggregate (page 834) 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 247).

```
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() (page 945) helper in the mongo (page 1066) shell, and would resemble the following:

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : "$tags",
     authors : { $addToSet : "$author" }
  } }
);
```

Changed in version 2.4: If an error occurs, the aggregate() (page 945) helper throws an exception. In previous versions, the helper returned a document with the error message and code, and ok status field not equal to 1, same as the aggregate (page 834) command.

For more aggregation documentation, please see:

- *Aggregation Framework* (page 247)
- *Aggregation Framework Reference* (page 263)
- *Aggregation Framework Examples* (page 253)

count

Definition

count

Counts the number of documents in a collection. Returns a document that contains this count and as well as the command status. count (page 834) has the following form:

```
{ count: <collection>, query: <query>, limit: <limit>, skip: <skip> }
```

`count` (page 834) has the following fields:

field string count The name of the collection to count.

field document query A query that selects which documents to count in a collection.

field integer limit The maximum number of matching documents to return.

field integer skip The number of matching documents to skip before returning results.

Note: MongoDB also provides the `count()` (page 978) and `db.collection.count()` (page 946) wrapper methods in the `mongo` (page 1066) shell.

Examples The following sections provide examples of the `count` (page 834) command.

Count all Documents The following operation counts the number of all documents in the `orders` collection:

```
db.runCommand( { count: 'orders' } )
```

In the result, the `n`, which represents the count, is `26`, and the command status `ok` is `1`:

```
{ "n" : 26, "ok" : 1 }
```

Count Documents with Specified Field Values The following operation returns a count of the documents in the `orders` collection where the value of the `ord_dt` field is greater than `Date('01/01/2012')`:

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } }
               } )
```

In the result, the `n`, which represents the count, is `13` and the command status `ok` is `1`:

```
{ "n" : 13, "ok" : 1 }
```

Skip Matching Documents The following operation returns a count of the documents in the `orders` collection where the value of the `ord_dt` field is greater than `Date('01/01/2012')` and skip the first `10` matching documents:

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } },
                 skip: 10 }  )
```

In the result, the `n`, which represents the count, is `3` and the command status `ok` is `1`:

```
{ "n" : 3, "ok" : 1 }
```

distinct

Definition

distinct

Finds the distinct values for a specified field across a single collection. `distinct` (page 835) returns a document that contains an array of the distinct values. The return document also contains a subdocument with query statistics and the query plan.

When possible, the `distinct` (page 835) command uses an index to find documents and return values.

The command takes the following form:

```
{ distinct: "<collection>", key: "<field>", query: <query> }
```

The command contains the following fields:

field string distinct The name of the collection to query for distinct values.

field string key The field to collect distinct values from.

field document query A query specification to limit the input documents in the *distinct* analysis.

Examples Return an array of the distinct values of the field `ord_dt` from all documents in the `orders` collection:

```
db.runCommand ( { distinct: "orders", key: "ord_dt" } )
```

Return an array of the distinct values of the field `sku` in the subdocument `item` from all documents in the `orders` collection:

```
db.runCommand ( { distinct: "orders", key: "item.sku" } )
```

Return an array of the distinct values of the field `ord_dt` from the documents in the `orders` collection where the `price` is greater than `10`:

```
db.runCommand ( { distinct: "orders",
                  key: "ord_dt",
                  query: { price: { $gt: 10 } }
                } )
```

Note: MongoDB also provides the shell wrapper method `db.collection.distinct()` (page 948) for the `distinct` (page 835) command. Additionally, many MongoDB *drivers* also provide a wrapper method. Refer to the specific driver documentation.

group

group

The `group` (page 836) command groups documents in a collection by the specified key and performs simple aggregation functions such as computing counts and sums. The command is analogous to a `SELECT ... GROUP BY` statement in SQL. The command returns a document with the grouped records as well as the command meta-data.

The `group` (page 836) command takes the following prototype form:

```
{ group: { ns: <namespace>,
           key: <key>,
           $reduce: <reduce function>,
           $keyf: <key function>,
           cond: <query>,
           finalize: <finalize function> } }
```

The command fields are as follows:

Fields

- **ns** – Specifies the collection from which to perform the group by operation.
- **key** – Specifies one or more document fields to group. Returns a "key object" for use as the grouping key.
- **$reduce** – Specifies an aggregation function that operates on the documents during the grouping operation, such as compute a sum or a count. The aggregation function takes two arguments: the current document and an aggregation result document for that group.
- **initial** – Initializes the aggregation result document.
- **$keyf** – Optional. Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use the `keyf` instead of `key` to group by calculated fields rather than existing document fields.
- **cond** – Optional. Specifies the selection criteria to determine which documents in the collection to process. If the `cond` field is omitted, the `db.collection.group()` (page 958) processes all the documents in the collection for the group operation.
- **finalize** – Optional. Specifies a function that runs each item in the result set before `db.collection.group()` (page 958) returns the final value. This function can either modify the result document or replace the result document as a whole.

Note: Unlike the `$keyf` and the `$reduce` fields that specify a function, the field name is `finalize` and not `$finalize`.

Warning:

- The `group` (page 836) command does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.
- The result set must fit within the *maximum BSON document size* (page 1139).
- Additionally, in version 2.2, the returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. For group by operations that results in more than 20,000 unique groupings, use `mapReduce` (page 840). Previous versions had a limit of 10,000 elements.
- Prior to 2.4, the `group` (page 836) command took the `mongod` (page 1049) instance's JavaScript lock which blocked all other JavaScript execution.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 840), `group` (page 836) commands, or `$where` (page 797) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
args MaxKey MinKey	assert() BinData() DBPointer() DBRef() doassert() emit() gc() HexData() hex_md5() isNumber() isObject() ISODate() isString()	Map() MD5() NumberInt() NumberLong() ObjectId() print() printjson() printjsononeline() sleep() Timestamp() tojson() tojsononeline() tojsonObject() UUID() version()

For the shell, MongoDB provides a wrapper method `db.collection.group()` (page 958); however, the `db.collection.group()` (page 958) method takes the `keyf` field and the `reduce` field whereas the `group` (page 836) command takes the `$keyf` field and the `$reduce` field.

JavaScript in MongoDB

Although `group` (page 836) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 575) in the language of the interacting application.

The following are examples of the `db.collection.group()` (page 958) method:

The examples assume an `orders` collection with documents of the following prototype:

```
{
  _id: ObjectId("5085a95c8fada716c89d0021"),
  ord_dt: ISODate("2012-07-01T04:00:00Z"),
  ship_dt: ISODate("2012-07-02T04:00:00Z"),
  item: { sku: "abc123",
          price: 1.99,
          uom: "pcs",
          qty: 25 }
}
```

- The following example groups by the `ord_dt` and `item.sku` fields those documents that have `ord_dt` greater than `01/01/2012`:

```
db.runCommand( { group:
                   {
                     ns: 'orders',
                     key: { ord_dt: 1, 'item.sku': 1 },
                     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                     $reduce: function ( curr, result ) { },
                     initial: { }
                   }
                } )
```

The result is a documents that contain the `retval` field which contains the group by records, the `count` field which contains the total number of documents grouped, the `keys` field which contains the number of unique groupings (i.e. number of elements in the `retval`), and the `ok` field which contains the command status:

```
{ "retval" :
      [ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123"},
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456"},
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123"},
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456"},
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123"},
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456"}
      ],
  "count" : 13,
  "keys" : 11,
  "ok" : 1 }
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

• The following example groups by the `ord_dt` and `item.sku` fields, those documents that have `ord_dt` greater than `01/01/2012` and calculates the sum of the `qty` field for each grouping:

```
db.runCommand( { group:
                   {
                     ns: 'orders',
                     key: { ord_dt: 1, 'item.sku': 1 },
                     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                     $reduce: function ( curr, result ) {
                                  result.total += curr.item.qty;
                              },
                     initial: { total : 0 }
                   }
                } )
```

The `retval` field of the returned document is an array of documents that contain the group by fields and the calculated aggregation field:

```
{ "retval" :
      [ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123", "total" : 10 },
        { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456", "total" : 10 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456", "total" : 15 },
        { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123", "total" : 20 },
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123", "total" : 45 },
        { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
        { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456", "total" : 25 }
```

```
    ],
  "count" : 13,
  "keys" : 11,
  "ok" : 1 }
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku, SUM(item_qty) as total
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

• The following example groups by the calculated `day_of_week` field, those documents that have `ord_dt` greater than `01/01/2012` and calculates the sum, count, and average of the `qty` field for each grouping:

```
db.runCommand( { group:
                   {
                     ns: 'orders',
                     $keyf: function(doc) {
                                return { day_of_week: doc.ord_dt.getDay() } ; },
                     cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                     $reduce: function ( curr, result ) {
                                 result.total += curr.item.qty;
                                 result.count++;
                             },
                     initial: { total : 0, count: 0 },
                     finalize: function(result) {
                                  var weekdays = [ "Sunday", "Monday", "Tuesday",
                                                   "Wednesday", "Thursday",
                                                   "Friday", "Saturday" ];

                                  result.day_of_week = weekdays[result.day_of_week];
                                  result.avg = Math.round(result.total / result.count);
                     }

                   }
                } )
```

The `retval` field of the returned document is an array of documents that contain the group by fields and the calculated aggregation field:

```
{ "retval" :
      [ { "day_of_week" : "Sunday", "total" : 70, "count" : 4, "avg" : 18 },
        { "day_of_week" : "Friday", "total" : 110, "count" : 6, "avg" : 18 },
        { "day_of_week" : "Tuesday", "total" : 70, "count" : 3, "avg" : 23 }

      ],
  "count" : 13,
  "keys" : 3,
  "ok" : 1 }
```

See also:

Aggregation Framework (page 247)

mapReduce

mapReduce

The mapReduce (page 840) command allows you to run *map-reduce* aggregation operations over a collection. The mapReduce (page 840) command has the following prototype form:

```
db.runCommand(
               {
                 mapReduce: <collection>,
                 map: <function>,
                 reduce: <function>,
                 out: <output>,
                 query: <document>,
                 sort: <document>,
                 limit: <number>,
                 finalize: <function>,
                 scope: <document>,
                 jsMode: <boolean>,
                 verbose: <boolean>
               }
             )
```

Pass the name of the collection to the `mapReduce` command (i.e. `<collection>`) to use as the source documents to perform the map reduce operation. The command also accepts the following parameters:

Parameters

- **map** – A JavaScript function that associates or "maps" a `value` with a `key` and emits the `key` and value `pair`.

  The `map` function processes every input document for the map-reduce operation. However, the `map` function can call emit any number of times, including 0, for each input document. The map-reduce operation groups the emitted `value` objects by the `key` and passes these groupings to the `reduce` function. See below for requirements for the `map` function.

- **reduce** – A JavaScript function that "reduces" to a single object all the `values` associated with a particular `key`.

  The `reduce` function accepts two arguments: `key` and `values`. The `values` argument is an array whose elements are the `value` objects that are "mapped" to the `key`. See below for requirements for the `reduce` function.

- **out** – New in version 1.8.

  Specifies the location of the result of the map-reduce operation. You can output to a collection, output to a collection with an action, or output inline. You may output to a collection when performing map reduce operations on the primary members of the set; on *secondary* members you may only use the `inline` output.

- **query** – Optional. Specifies the selection criteria using *query operators* (page 785) for determining the documents input to the `map` function.

- **sort** – Optional. Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.

- **limit** – Optional. Specifies a maximum number of documents to return from the collection.

- **finalize** – Optional. A JavaScript function that follows the `reduce` method and modifies the output.

  The `finalize` function receives two arguments: `key` and `reducedValue`. The `reducedValue` is the value returned from the `reduce` function for the `key`.

- **scope** (*document*) – Optional. Specifies global variables that are accessible in the `map` , `reduce` and the `finalize` functions.
- **jsMode** (*Boolean*) – New in version 2.0.

  Optional. Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions.

  If `false`:

  – Internally, MongoDB converts the JavaScript objects emitted by the `map` function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the `reduce` function.

  – The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets.

  If `true`:

  – Internally, the JavaScript objects emitted during `map` function remain as JavaScript objects. There is no need to convert the objects for the `reduce` function, which can result in faster execution.

  – You can only use `jsMode` for result sets with fewer than 500,000 distinct `key` arguments to the mapper's `emit()` function.

  The `jsMode` defaults to false.

- **verbose** (*Boolean*) – Optional. Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

The following is a prototype usage of the `mapReduce` (page 840) command:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
               {
                 mapReduce: 'orders',
                 map: mapFunction,
                 reduce: reduceFunction,
                 out: { merge: 'map_reduce_results', db: 'test' },
                 query: { ord_date: { $gt: new Date('01/01/2012') } }
               }
             )
```

JavaScript in MongoDB

Although `mapReduce` (page 840) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 575) in the language of the interacting application.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 840), `group` (page 836) commands, or `$where` (page 797) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	
args MaxKey MinKey	assert() BinData() DBPointer() DBRef() doassert() emit() gc() HexData() hex_md5() isNumber() isObject() ISODate() isString()	Map() MD5() NumberInt() NumberLong() ObjectId() print() printjson() printjsononeline() sleep() Timestamp() tojson() tojsononeline() tojsonObject() UUID() version()

Requirements for the `map` Function The `map` function has the following prototype:

```
function() {
   ...
   emit(key, value);
}
```

The `map` function exhibits the following behaviors:

- In the `map` function, reference the current document as `this` within the function.
- The `map` function should *not* access the database for any reason.
- The `map` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `emit(key,value)` function associates the `key` with a `value`.
  - A single emit can only hold half of MongoDB's *maximum BSON document size* (page 1139).
  - The `map` function can call `emit(key,value)` any number of times, including 0, per each input document.

    The following `map` function may call `emit(key,value)` either 0 or 1 times depending on the value of the input document's `status` field:

    ```
    function() {
        if (this.status == 'A')
            emit(this.cust_id, 1);
    }
    ```

    The following `map` function may call `emit(key,value)` multiple times depending on the number of elements in the input document's `items` field:

    ```
    function() {
        this.items.forEach(function(item){ emit(item.sku, 1); });
    }
    ```

- The `map` function can access the variables defined in the `scope` parameter.

Requirements for the `reduce` Function The `reduce` function has the following prototype:

```
function(key, values) {
   ...
   return result;
}
```

The `reduce` function exhibits the following behaviors:

- The `reduce` function should *not* access the database, even to perform read operations.
- The `reduce` function should *not* affect the outside system.
- MongoDB will **not** call the `reduce` function for a key that has only a single value.
- MongoDB can invoke the `reduce` function more than once for the same key. In this case, the previous output from the `reduce` function for that key will become one of the input values to the next `reduce` function invocation for that key.
- The `reduce` function can access the variables defined in the `scope` parameter.

Because it is possible to invoke the `reduce` function more than once for the same key, the following properties need to be true:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function to ensure that the following operations is true:

  ```
  reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
  ```

- the `reduce` function must be *idempotent*. Ensure that the following statement is true:

  ```
  reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
  ```

- the order of the elements in the `valuesArray` should not affect the output of the `reduce` function, so that the following statement is true:

  ```
  reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
  ```

`out` Options You can specify the following options for the `out` parameter:

Output to a Collection

```
out: <collectionName>
```

Output to a Collection with an Action This option is only available when passing `out` a collection that already exists. This option is not available on secondary members of replica sets.

```
out: { <action>: <collectionName>
        [, db: <dbName>]
        [, sharded: <boolean> ]
        [, nonAtomic: <boolean> ] }
```

When you output to a collection with an action, the `out` has the following parameters:

- `<action>`: Specify one of the following actions:

  - `replace`

    Replace the contents of the `<collectionName>` if the collection with the `<collectionName>` exists.

  - `merge`

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, *overwrite* that existing document.

  - `reduce`

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, apply the `reduce` function to both the new and the existing documents and overwrite the existing document with the result.

- `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* you have enabled sharding on output database, the map-reduce operation will shard the output collection using the `_id` field as the shard key.

- `nonAtomic`:

  New in version 2.2.

  Optional. Specify output operation as non-atomic and is valid *only* for `merge` and `reduce` output modes which may take minutes to execute.

  If `nonAtomic` is `true`, the post-processing step will prevent MongoDB from locking the database; however, other clients will be able to read intermediate states of the output collection. Otherwise the map reduce operation must lock the database during post-processing.

Output Inline Perform the map-reduce operation in memory and return the result. This option is the only available option for `out` on secondary members of replica sets.

```
out: { inline: 1 }
```

The result must fit within the *maximum size of a BSON document* (page 1139).

Requirements for the `finalize` Function The `finalize` function has the following prototype:

```
function(key, reducedValue) {
   ...
   return modifiedObject;
}
```

The `finalize` function receives as its arguments a `key` value and the `reducedValue` from the `reduce` function. Be aware that:

- The `finalize` function should *not* access the database for any reason.
- The `finalize` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `finalize` function can access the variables defined in the `scope` parameter.

Examples In the mongo (page 1066) shell, the db.collection.mapReduce() (page 963) method is a wrapper around the mapReduce (page 840) command. The following examples use the db.collection.mapReduce() (page 963) method:

Consider the following map-reduce operations on a collection orders that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer Perform map-reduce operation on the orders collection to group by the cust_id, and for each cust_id, calculate the sum of the price for each cust_id:

1. Define the map function to process each input document:

   - In the function, this refers to the document that the map-reduce operation is processing.
   - The function maps the price to the cust_id for each document and emits the cust_id and price pair.

   ```
   var mapFunction1 = function() {
                          emit(this.cust_id, this.price);
                      };
   ```

2. Define the corresponding reduce function with two arguments keyCustId and valuesPrices:

   - The valuesPrices is an array whose elements are the price values emitted by the map function and grouped by keyCustId.
   - The function reduces the valuesPrice array to the sum of its elements.

   ```
   var reduceFunction1 = function(keyCustId, valuesPrices) {
                             return Array.sum(valuesPrices);
                         };
   ```

3. Perform the map-reduce on all documents in the orders collection using the mapFunction1 map function and the reduceFunction1 reduce function.

   ```
   db.orders.mapReduce(
                        mapFunction1,
                        reduceFunction1,
                        { out: "map_reduce_example" }
                      )
   ```

   This operation outputs the results to a collection named map_reduce_example. If the map_reduce_example collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item In this example you will perform a map-reduce operation on the orders collection, for all documents that have an ord_date value greater than

`01/01/2012`. The operation groups by the `item.sku` field, and for each `sku` calculates the number of orders and the total quantity ordered. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

   ```
   var mapFunction2 = function() {
                          for (var idx = 0; idx < this.items.length; idx++) {
                              var key = this.items[idx].sku;
                              var value = {
                                            count: 1,
                                            qty: this.items[idx].qty
                                          };
                              emit(key, value);
                          }
                       };
   ```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

   ```
   var reduceFunction2 = function(keySKU, countObjVals) {
                        reducedVal = { count: 0, qty: 0 };

                        for (var idx = 0; idx < countObjVals.length; idx++) {
                            reducedVal.count += countObjVals[idx].count;
                            reducedVal.qty += countObjVals[idx].qty;
                        }

                        return reducedVal;
                     };
   ```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

   ```
   var finalizeFunction2 = function (key, reducedVal) {

                          reducedVal.avg = reducedVal.qty/reducedVal.count;

                          return reducedVal;

                       };
   ```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

   ```
   db.orders.mapReduce( mapFunction2,
                        reduceFunction2,
                        {
   ```

```
            out: { merge: "map_reduce_example" },
            query: { ord_date:
                       { $gt: new Date('01/01/2012') }
                   },
            finalize: finalizeFunction2
          }
        )
```

This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

For more information and examples, see the *Map-Reduce* (page 291) page and *Perform Incremental Map-Reduce* (page 293).

See also:

- *Troubleshoot the Map Function* (page 297)
- *Troubleshoot the Reduce Function* (page 298)
- `db.collection.mapReduce()` (page 963)
- *Aggregation Framework* (page 247)

Comparison The following table provides a brief overview of the features of the commands.

	`aggregate` (page 834)	`mapReduce` (page 840)	`group` (page 836)
Description	New in version 2.2. Designed with specific goals of improving performance and usability for aggregation tasks. Uses a "pipeline" approach where objects are transformed as they pass through a series of pipeline operators such as `$group` (page 269), `$match` (page 266), and `$sort` (page 270). See *Aggregation Framework Reference* (page 263) for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the `aggregate` (page 834) command and has less functionality than the `mapReduce` (page 840) command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See *Map-Reduce Examples* (page 291) and *Perform Incremental Map-Reduce* (page 293).	Can either group by existing fields or with a custom `keyf` JavaScript function, can group by calculated fields. See `group` (page 836) for information and example using the `keyf` function.
Flexibility	Limited to the operators and expressions supported by the Aggregation Framework. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the `$project` (page 264) pipeline operator. See `$project` (page 264) for more information as well as *Aggregation Framework Reference* (page 263) for more information on all the available pipeline operators.	Custom `map`, `reduce` and `finalize` JavaScript functions offer flexibility to aggregation logic. See `mapReduce` (page 840) for details and restrictions on the functions.	Custom `reduce` and `finalize` JavaScript functions offer flexibility to grouping logic. See `group` (page 836) for details and restrictions on these functions.
Output Results	Returns results inline. The result is subject to the *BSON Document size* (page 1139) limit.	Returns results in various options (inline, new collection, merge, replace, reduce). See `mapReduce` (page 840) for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the *maximum BSON document size limit* (page 1139). Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections.	Does **not** support sharded collection.
Notes		Prior to 2.4, JavaScript code executed in a single thread.	Prior to 2.4, JavaScript code executed in a single thread.
More Informa-	See *Aggregation Framework* (page 247) and `aggregate` (page 834).	See *Map-Reduce* (page 291) and `mapReduce` (page 840).	See `group` (page 836).

Geospatial Commands

Geospatial Commands

Name	Description
geoNear (page 850)	Performs a geospatial query that returns the 100 documents closest to a given point.
geoSearch (page 851)	Performs a geospatial query that uses MongoDB's *haystack index* functionality.
geoWalk (page 851)	An internal command to support geospatial queries.

geoNear

geoNear

The geoNear (page 850) command specifies a point for which a geospatial query returns the 100 closest documents. The geoNear (page 850) command provides an alternative to the $near (page 801) operator. In addition to the functionality of $near (page 801), geoNear (page 850) returns additional diagnostic information.

The geoNear (page 850) command can use either a *GeoJSON* point or *legacy coordinate pairs*.

To query a *2dsphere* (page 344) index, use the following syntax:

```
db.runCommand( { geoNear : <collection> ,
                 near : { type : "Point" ,
                          coordinates: [ <coordinates> ] } ,
                 spherical : true } )
```

To query a *2d* (page 341) index, use:

```
{ geoNear : <collection> , near : [ <coordinates> ] }
```

The geoNear (page 850) command provides the following options. Specify all distances in the same units as the document coordinate system:

Fields

- **near** – Can use either a GeoJSON point or legacy points, as shown above.
- **limit** – Optional. Specifies the maximum number of documents to return. The default value is 100. See also the `num` option.
- **num** – Optional. Synonym for the `limit` option. If both `num` and `limit` are included, the `num` value overrides the `limit` value.
- **maxDistance** – Optional. Limits the results to those falling within a given distance of the center coordinate. For *GeoJSON* data distance is in meters. For *grid coordinate* (page 341) data distance is in radians.
- **query** – Optional. Further narrows the results using any standard MongoDB query operator or selection. See db.collection.find() (page 951) and *Query, Update and Projection Operators* (page 785) 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 (page 850). 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` (page 807) for more information on this option

geoSearch

`geoSearch`

The `geoSearch` (page 851) 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` (page 851) command limits results to 50 documents.

geoWalk

`geoWalk`

`geoWalk` (page 851) is an internal command.

Query and Write Operation Commands

Query and Write Operation Commands

Name	Description
`findAndModify` (page 851)	Returns and modifies a single document.
`text` (page 856)	Performs a text search.
`getLastError` (page 861)	Returns the success status of the last operation.
`getPrevError` (page 862)	Returns status document containing all errors since the last `resetError` (page 862) command.
`resetError` (page 862)	Resets the last error status.
`eval` (page 862)	Runs a JavaScript function on the database server.

findAndModify

`findAndModify`

The `findAndModify` (page 851) 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: <string>,
  query: <document>,
  sort: <document>,
  remove: <boolean>,
  update: <document>,
  new: <boolean>,
  fields: <document>,
```

```
  upsert: <boolean>
}
```

The findAndModify (page 851) command takes the following fields:

Fields

- **findAndModify** (*string*) – Required. The collection against which to run the command.
- **query** (*document*) – Optional. Specifies the selection criteria for the modification. The query field employs the same *query selectors* (page 785) as used in the db.collection.find() (page 951) method. Although the query may match multiple documents, findAndModify (page 851) will only select one document to modify.
- **sort** (*document*) – Optional. Determines which document the operation will modify if the query selects multiple documents. findAndModify (page 851) will modify the first document in the sort order specified by this argument.
- **remove** (*boolean*) – Must specify either the remove or the update field in the findAndModify (page 851) command. When true, removes the selected document. The default is false.
- **update** (*document*) – Must specify either the remove or the update field in the findAndModify (page 851) command. The update field employs the same *update operators* (page 810) or field: value specifications to modify the selected document.
- **new** (*boolean*) – Optional. When true, returns the modified document rather than the original. The findAndModify (page 851) method ignores the new option for remove operations. The default is false.
- **fields** (*document*) – Optional. A subset of fields to return. The fields document specifies an inclusion of a field with 1, as in the following:

  ```
  fields: { <field1>: 1, <field2>: 1, ... }
  ```

  See *projection* (page 45).
- **upsert** (*boolean*) – Optional. Used in conjunction with the update field. When true, the findAndModify (page 851) command creates a new document if the query returns no documents. The default is false.

The findAndModify (page 851) command returns a document, similar to the following:

```
{
   lastErrorObject: {
                          updatedExisting: <boolean>,
                          upserted: <boolean>,
                          n: <num>,
                          connectionId: <num>,
                          err: <string>,
                          ok: <num>
   }
   value: <document>,
   ok: <num>
}
```

The return document contains the following fields:

- The lastErrorObject field that returns the details of the command:
  - The updatedExisting field **only** appears if the command is either an update or an upsert.

–The `upserted` field **only** appears if the command is an `upsert`.

•The `value` field that returns either:

–the original (i.e. pre-modification) document if `new` is false, or

–the modified or inserted document if `new: true`.

•The `ok` field that returns the status of the command.

Note: If the `findAndModify` (page 851) finds no matching document, then:

•for `update` or `remove` operations, `lastErrorObject` does not appear in the return document and the `value` field holds a `null`.

```
{ "value" : null, "ok" : 1 }
```

•for an `upsert` operation that performs an insert, when `new` is `false`, **and** includes a `sort` option, the return document has `lastErrorObject`, `value`, and `ok` fields, but the `value` field holds an empty document `{}`.

•for an `upsert` that performs an insert, when `new` is `false` **without** a specified `sort` the return document has `lastErrorObject`, `value`, and `ok` fields, but the `value` field holds a `null`.

Changed in version 2.2: Previously, the command returned an empty document (e.g. `{}`) in the `value` field. See *the 2.2 release notes* (page 1192) for more information.

Consider the following examples:

•The following command updates an existing document in the `people` collection where the document matches the `query` criteria:

```
db.runCommand(
                {
                  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` (page 786) 10.

2.The `sort` orders the results of the query in ascending order. If multiple documents meet the `query` condition, the command will select for modification the first document as ordered by this `sort`.

3.The `update` `increments` (page 810) the value of the `score` field by 1.

4.The command returns a document with the following fields:

–The `lastErrorObject` field that contains the details of the command, including the field `updatedExisting` which is `true`, and

–The `value` field that contains the original (i.e. pre-modification) document selected for this update:

```
{
  "lastErrorObject" : {
```

```
        "updatedExisting" : true,
        "n" : 1,
        "connectionId" : 1,
        "err" : null,
        "ok" : 1
    },
    "value" : {
        "_id" : ObjectId("50f1d54e9beb36a0f45c6452"),
        "name" : "Tom",
        "state" : "active",
        "rating" : 100,
        "score" : 5
    },
    "ok" : 1
}
```

To return the modified document in the `value` field, add the `new:true` option to the command.

If no document match the `query` condition, the command returns a document that contains `null` in the `value` field:

```
{ "value" : null, "ok" : 1 }
```

The `mongo` (page 1066) shell and many *drivers* provide a `findAndModify()` (page 952) helper method. Using the shell helper, this previous operation can take the following form:

```
db.people.findAndModify( {
   query: { name: "Tom", state: "active", rating: { $gt: 10 } },
   sort: { rating: 1 },
   update: { $inc: { score: 1 } }
} );
```

However, the `findAndModify()` (page 952) shell helper method returns just the unmodified document, or the modified document when `new` is `true`.

```
{
   "_id" : ObjectId("50f1d54e9beb36a0f45c6452"),
   "name" : "Tom",
   "state" : "active",
   "rating" : 100,
   "score" : 5
}
```

•The following `findAndModify` (page 851) command includes the `upsert: true` option to insert a new document if no document matches the `query` condition:

```
db.runCommand(
               {
                 findAndModify: "people",
                 query: { name: "Gus", state: "active", rating: 100 },
                 sort: { rating: 1 },
                 update: { $inc: { score: 1 } },
                 upsert: true
               }
             )
```

If the command does **not** find a matching document, the command performs an upsert and returns a document with the following fields:

– The `lastErrorObject` field that contains the details of the command, including the field `upserted` that contains the `ObjectId` of the newly inserted document, and

– The `value` field that contains an empty document `{}` as the original document because the command included the `sort` option:

```
{
  "lastErrorObject" : {
     "updatedExisting" : false,
     "upserted" : ObjectId("50f2329d0092b46dae1dc98e"),
     "n" : 1,
     "connectionId" : 1,
     "err" : null,
     "ok" : 1
  },
  "value" : {

  },
  "ok" : 1
}
```

If the command did **not** include the `sort` option, the `value` field would contain `null`:

```
{
  "value" : null,
  "lastErrorObject" : {
     "updatedExisting" : false,
     "n" : 1,
     "upserted" : ObjectId("5102f7540cb5c8be998c2e99")
  },
  "ok" : 1
}
```

• The following `findAndModify` (page 851) command includes both `upsert: true` option and the `new:true` option to return the newly inserted document in the `value` field if a document matching the `query` is not found:

```
db.runCommand(
               {
                 findAndModify: "people",
                 query: { name: "Pascal", state: "active", rating: 25 },
                 sort: { rating: 1 },
                 update: { $inc: { score: 1 } },
                 upsert: true,
                 new: true
               }
             )
```

The command returns the newly inserted document in the `value` field:

```
{
  "lastErrorObject" : {
     "updatedExisting" : false,
     "upserted" : ObjectId("50f47909444c11ac2448a5ce"),
     "n" : 1,
     "connectionId" : 1,
     "err" : null,
     "ok" : 1
  },
  "value" : {
```

```
            "_id" : ObjectId("50f47909444c11ac2448a5ce"),
            "name" : "Pascal",
            "rating" : 25,
            "score" : 1,
            "state" : "active"
        },
        "ok" : 1
    }
```

When the `findAndModify` (page 851) command includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients issue the `findAndModify` (page 851) command and these commands complete the `find` phase before any one starts the `modify` phase, these commands could insert the same document.

Consider an example where no document with the name `Andy` exists and multiple clients issue the following command:

```
db.runCommand(
                {
                  findAndModify: "people",
                  query: { name: "Andy" },
                  sort: { rating: 1 },
                  update: { $inc: { score: 1 } },
                  upsert: true
                }
              )
```

If all the commands finish the `query` phase before any command starts the `modify` phase, **and** there is no unique index on the `name` field, the commands may all perform an upsert. To prevent this condition, create a *unique index* (page 314) on the `name` field. With the unique index in place, then the multiple `findAndModify` (page 851) commands would observe one of the following behaviors:

- Exactly one `findAndModify` (page 851) would successfully insert a new document.
- Zero or more `findAndModify` (page 851) commands would update the newly inserted document.
- Zero or more `findAndModify` (page 851) commands would fail when they attempted to insert a duplicate. If the command fails due to a unique index constraint violation, you can retry the command. Absent a delete of the document, the retry should not fail.

Warning: When using `findAndModify` (page 851) in a *sharded* environment, the `query` must contain the *shard key* for all operations against the shard cluster. `findAndModify` (page 851) operations issued against `mongos` (page 1061) 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()` (page 974) operations.

text

Definition

`text`

New in version 2.4.

Searches text content stored in the *text index* (page 318). The `text` (page 856) command is **case-insensitive**.

The `text` (page 856) command returns all documents that contain any of the terms; i.e. it performs a logical `OR` search. By default, the command limits the matches to the top 100 scoring documents, in descending score order, but you can specify a different limit.

The `text` (page 856) command has the following syntax:

```
db.collection.runCommand( "text", { search: <string>,
                                    filter: <document>,
                                    project: <document>,
                                    limit: <number>,
                                    language: <string> } )
```

The `text` (page 856) command has the following parameters:

field string search A string of terms that MongoDB parses and uses to query the `text` index. Enclose the string of terms in escaped double quotes to match on the phrase. For further information on the `search` field syntax, see *The search Field* (page 858).

field document filter A *query document* (page 64) to further limit the results of the query using another database field. Use any valid MongoDB query in the filter document, except if the index includes an ascending or descending index field as a prefix. If the index includes an ascending or descending index field as a prefix, the `filter` is required and the `filter` query must be an equality match.

field document project Limits the fields returned by the query to only those specified. By default, the `_id` field returns as part of the result set, *unless* you explicitly exclude the field in the project document.

field number limit The maximum number of documents to include in the response. The `text` (page 856) command sorts the results before applying the `limit`. The default limit is 100.

field string language The language that determines the list of stop words for the search and the rules for the stemmer and tokenizer. If not specified, the search uses the default language of the index. For supported languages, see *Text Search Languages* (page 860). Specify the language in **lowercase**.

Returns The `text` (page 856) command returns a document that contains a field `results` that contains an array of the highest scoring documents, in descending order by score. See *Output* (page 859) for details.

> **Warning:** The complete results of the `text` (page 856) command must fit within the `BSON Document Size` (page 1139). Otherwise, the command will limit the results to fit within the `BSON Document Size` (page 1139). Use the `limit` and the `project` parameters with the `text` (page 856) command to limit the size of the result set.

Note:

- If the `search` string includes phrases, the search performs an `AND` with any other terms in the search string; e.g. search for `"\"twinkle twinkle\" little star"` searches for `"twinkle twinkle"` **and** (`"little"` **or** `"star"`).
- `text` (page 856) adds all negations to the query with the logical `AND` operator.
- The `text` (page 856) command ignores stop words for the search language, such as `the` and `and` in English.
- The `text` (page 856) command matches on the complete *stemmed* word. So if a document field contains the word `blueberry`, a search on the term `blue` will not match. However, `blueberry` or `blueberries` will match.

The `search` Field The `search` field takes a string of terms that MongoDB parses and uses to query the `text` index. Enclose the string of terms in escaped double quotes to match on the phrase. Additionally, the `text` (page 856) command treats most punctuation as delimiters, except when a hyphen - negates terms.

Prefixing a word with a hyphen sign (-) negates a word:

- The negated word excludes documents that contain the negated word from the result set.
- A search string that only contains negated words returns **no** match.
- A hyphenated word, such as `pre-market`, is not a negation. The text command treats the hyphen as a delimiter.

Examples The following examples assume a collection `articles` that has a text index on the field `subject`:

```
db.articles.ensureIndex( { subject: "text" } )
```

Search for a Single Word

```
db.articles.runCommand( "text", { search: "coffee" } )
```

This query returns documents that contain the word `coffee`, case-insensitive, in the indexed `subject` field.

Search for Multiple Words The following command searches for `bake` or `coffee` or `cake`:

```
db.articles.runCommand( "text", { search: "bake coffee cake" } )
```

This query returns documents that contain either `bake` **or** `coffee` **or** `cake` in the indexed `subject` field.

Search for a Phrase

```
db.articles.runCommand( "text", { search: "\"bake coffee cake\"" } )
```

This query returns documents that contain the phrase `bake coffee cake`.

Exclude a Term from the Result Set Use the hyphen (-) as a prefix to exclude documents that contain a term. Search for documents that contain the words `bake` or `coffee` but do **not** contain `cake`:

```
db.articles.runCommand( "text", { search: "bake coffee -cake" } )
```

Search with Additional Query Conditions Use the `filter` option to include additional query conditions.

Search for a single word `coffee` with an additional filter on the `about` field, but limit the results to 2 documents with the highest score and return only the `subject` field in the matching documents:

```
db.articles.runCommand( "text", {
                                    search: "coffee",
                                    filter: { about: /desserts/ },
                                    limit: 2,
                                    project: { subject: 1, _id: 0 }
                                  }
                      )
```

- The `filter` *query document* (page 64) may use any of the available *query operators* (page 785).

- Because the `_id` field is implicitly included, in order to return **only** the `subject` field, you must explicitly exclude (`0`) the `_id` field. Within the `project` document, you cannot mix inclusions (i.e. `<fieldA>: 1`) and exclusions (i.e. `<fieldB>: 0`), except for the `_id` field.

Search a Different Language Use the `language` option to specify Spanish as the language that determines the list of stop words and the rules for the stemmer and tokenizer:

```
db.articles.runCommand( "text", {
                                    search: "leche",
                                    language: "spanish"
                                }
                      )
```

See *Text Search Languages* (page 860) for the supported languages.

Important: Specify the language in **lowercase**.

Output The following is an example document returned by the `text` (page 856) command:

```
{
   "queryDebugString" : "tomorrow||||||",
   "language" : "english",
   "results" : [
      {
         "score" : 1.3125,
         "obj": {
                  "_id" : ObjectId("50ecef5f8abea0fda30ceab3"),
                  "quote" : "tomorrow, and tomorrow, and tomorrow, creeps in this petty pace",
                  "related_quotes" : [
                                       "is this a dagger which I see before me",
                                       "the handle toward my hand?"
                                     ],
                  "src" : {
                             "title" : "Macbeth",
                             "from" : "Act V, Scene V"
                          },
                  "speaker" : "macbeth"
                }
      }
   ],
   "stats" : {
               "nscanned" : 1,
               "nscannedObjects" : 0,
               "n" : 1,
               "nfound" : 1,
               "timeMicros" : 163
            },
   "ok" : 1
}
```

The `text` (page 856) command returns the following data:

`text.`**`queryDebugString`**
For internal use only.

`text.`**`language`**
The `language` (page 859) field returns the language used for the text search. This language determines the list

of stop words and the rules for the stemmer and tokenizer.

`text.results`
The `results` (page 860) field returns an array of result documents that contain the information on the matching documents. The result documents are ordered by the `score` (page 860). Each result document contains:

`text.results.obj`
The `obj` (page 860) field returns the actual document from the collection that contained the stemmed term or terms.

`text.results.score`
The `score` (page 860) field for the document that contained the stemmed term or terms. The `score` (page 860) field signifies how well the document matched the stemmed term or terms. See *Control Results of Text Search with Weights* (page 357) for how you can adjust the scores for the matching words.

`text.stats`
The `stats` (page 860) field returns a document that contains the query execution statistics. The `stats` (page 860) field contains:

`text.stats.nscanned`
The `nscanned` (page 860) field returns the total number of index entries scanned.

`text.stats.nscannedObjects`
The `nscannedObjects` (page 860) field returns the total number of documents scanned.

`text.stats.n`
The `n` (page 860) field returns the number of elements in the `results` (page 860) array. This number may be less than the total number of matching documents, i.e. `nfound` (page 860), if the full result exceeds the `BSON Document Size` (page 1139).

`text.stats.nfound`
The `nfound` (page 860) field returns the total number number of documents that match. This number may be greater than the size of the `results` (page 860) array, i.e. `n` (page 860), if the result set exceeds the `BSON Document Size` (page 1139).

`text.stats.timeMicros`
The `timeMicros` (page 860) field returns the time in microseconds for the search.

`text.ok`
The `ok` (page 860) returns the status of the `text` (page 856) command.

Text Search Languages The *text index* (page 318) and the `text` (page 856) command support the following languages:

- `danish`
- `dutch`
- `english`
- `finnish`
- `french`
- `german`
- `hungarian`
- `italian`
- `norwegian`
- `portuguese`

- romanian
- russian
- spanish
- swedish
- turkish

Note: If you specify a language value of "none", then the text search has no list of stop words, and the text search does not stem or tokenize the search terms.

getLastError

Definition

getLastError

Returns the error status of the last operation on the *current connection*. Clients typically use getLastError (page 861) in combination with write operations to ensure that the write succeeds.

field Boolean j If true, wait for the next journal commit before returning, rather than waitng for a full disk flush. If mongod (page 1049) does not have journaling enabled, this option has no effect. If this option is enabled for a write operation, mongod (page 1049) will wait *no more* than 1/3 of the current journalCommitInterval (page 1119) before writing data to the journal.

field integer,string w When running with replication, this is the number of servers to replicate 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 (page 861) to block indefinitely.

field Boolean fsync If true, wait for mongod (page 1049) 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.

field integer wtimeout Milliseconds. Specify a value in milliseconds to control how long to wait for write propagation to complete. If replication does not complete in the given timeframe, the getLastError (page 861) command will return with an error status.

The following is the prototype form:

```
{ getLastError: 1 }
```

See also:

Write Concern (page 395) and *Replica Acknowledged* (page 396).

Examples

Confirm Replication to Two Replica Set Members The following example ensures the operation has replicated to two members (the primary and one other member):

```
db.runCommand( { getLastError: 1, w: 2 } )
```

Confirm Replication to a Majority of a Replica Set The following example ensures the write operation has replicated to a majority of the configured members of the set.

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

Set a Timeout for a `getLastError` Response Unless you specify a timeout, a `getLastError` (page 861) command may block forever if MongoDB cannot satisfy the requested write concern. To specify a timeout of 5000 milliseconds, use an invocation that resembles the following:

```
db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )
```

getPrevError

getPrevError

The `getPrevError` (page 862) command returns the errors since the last `resetError` (page 862) command.

See also:

`db.getPrevError()` (page 1006)

resetError

resetError

The `resetError` (page 862) command resets the last error status.

See also:

`db.resetError()` (page 1011)

eval

eval

The `eval` (page 862) command evaluates JavaScript functions on the database server and has the following form:

```
{
  eval: <function>,
  args: [ <arg1>, <arg2> ... ],
  nolock: <boolean>
}
```

The command contains the following fields:

field function eval A JavaScript function.

field array args An array of arguments to pass to the JavaScript function. Omit if the function does not take arguments.

field boolean nolock By default, `eval` (page 862) takes a global write lock before evaluating the JavaScript function. As a result, `eval` (page 862) blocks all other read and write operations to the database while the `eval` (page 862) operation runs. Set `nolock` to `true` on the `eval` (page 862) command to prevent the `eval` (page 862) command from taking the global write lock before evaluating the JavaScript. `nolock` does not impact whether operations within the JavaScript code itself takes a write lock.

JavaScript in MongoDB

Although `eval` (page 862) uses JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 575) in the language of the interacting application.

The following example uses eval (page 862) to perform an increment and calculate the average on the server:

```
db.runCommand( {
      eval: function(name, incAmount) {
               var doc = db.myCollection.findOne( { name : name } );

               doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

               doc.num++;
               doc.total += incAmount;
               doc.avg = doc.total / doc.num;

               db.myCollection.save( doc );
               return doc;
            },
      args: [ "eliot", 5 ]
   }
);
```

The db in the function refers to the current database.

The mongo (page 1066) shell provides a helper method db.eval() (page 1002) [2], so you can express the above as follows:

```
db.eval( function(name, incAmount) {
            var doc = db.myCollection.findOne( { name : name } );

            doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

            doc.num++;
            doc.total += incAmount;
            doc.avg = doc.total / doc.num;

            db.myCollection.save( doc );
            return doc;
         },
         "eliot", 5 );
```

If you want to use the server's interpreter, you must run eval (page 862). Otherwise, the mongo (page 1066) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, eval (page 862) throws an exception. The following invalid function uses the variable x without declaring it as an argument:

```
db.runCommand(
               {
                 eval: function() { return x + x; },
                 args: [ 3 ]
               }
             )
```

The statement will result in the following exception:

```
{
   "errmsg" : "exception: JavaScript execution failed: ReferenceError: x is not defined near '{
```

[2] The helper db.eval() (page 1002) in the mongo (page 1066) shell wraps the eval (page 862) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: db.eval() (page 1002) method does not support the nolock option.

```
    "code" : 16722,
    "ok" : 0
}
```

> **Warning:**
>
> - By default, `eval` (page 862) takes a global write lock before evaluating the JavaScript function. As a result, `eval` (page 862) blocks all other read and write operations to the database while the `eval` (page 862) operation runs. Set `nolock` to `true` on the `eval` (page 862) command to prevent the `eval` (page 862) command from taking the global write lock before evaluating the JavaScript. `nolock` does not impact whether operations within the JavaScript code itself takes a write lock.
> - Do not use `eval` (page 862) for long running operations as `eval` (page 862) blocks all other operations. Consider using *other server side code execution options* (page 581).
> - You can not use `eval` (page 862) with *sharded* data. In general, you should avoid using `eval` (page 862) in *sharded cluster*; nevertheless, it is possible to use `eval` (page 862) with non-sharded collections and databases stored in a *sharded cluster*.
> - With `authentication` (page 1118) enabled, `eval` (page 862) will fail during the operation if you do not have the permission to perform a specified task.
>
>   Changed in version 2.4: You must have full admin access to run.

Changed in version 2.4: The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, `eval` (page 862) executed in a single thread.

See also:

Server-side JavaScript (page 581)

55.2.2 Database Operations

Replication Commands

Replication Commands

Name	Description
`replSetFreeze` (page 865)	Prevents the current member from seeking election as *primary* for a period of time.
`replSetGetStatus` (page 865)	Returns a document that reports on the status of the replica set.
`replSetInitiate` (page 866)	Initializes a new replica set.
`replSetMaintenance` (page 867)	Enables or disables a maintenance mode, which puts a *secondary* node in a `RECOVERING` state.
`replSetReconfig` (page 868)	Applies a new configuration to an existing replica set.
`replSetStepDown` (page 868)	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
`replSetSyncFrom` (page 869)	Explicitly override the default logic for selecting a member to replicate from.
`resync` (page 870)	Forces a `mongod` (page 1049) to re-synchronize from the *master*. For master-slave replication only.
`applyOps` (page 870)	Internal command that applies *oplog* entries to the current data set.
`isMaster` (page 871)	Displays information about this member's role in the replica set, including whether it is the master.
`getoptime` (page 872)	Internal command to support replication, returns the optime.

replSetFreeze

replSetFreeze

The `replSetFreeze` (page 865) command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the `replSetStepDown` (page 868) command to make a different node in the replica set a primary.

The `replSetFreeze` (page 865) 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` (page 1049) process also unfreezes a replica set member.

`replSetFreeze` (page 865) is an administrative command, and you must issue it against the *admin database*.

replSetGetStatus

Definition

replSetGetStatus

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

The value specified does not affect the output of the command. Data provided by this command derives from data included in heartbeats sent to the current instance by other members of the replica set. Because of the frequency of heartbeats, these data can be several seconds out of date.

You can also access this functionality through the `rs.status()` (page 1017) helper in the `mongo` (page 1066) shell.

The `mongod` (page 1049) must have replication enabled and be a member of a replica set for the for `replSetGetStatus` (page 865) to return successfully.

Output

replSetGetStatus.**set**

The `set` value is the name of the replica set, configured in the `replSet` (page 1124) setting. This is the same value as `_id` (page 474) in `rs.conf()` (page 1015).

replSetGetStatus.**date**

The value of the `date` field is an *ISODate* of the current time, according to the current server. Compare this to the value of the `lastHeartbeat` (page 866) to find the operational lag between the current host and the other hosts in the set.

replSetGetStatus.**myState**

The value of `myState` (page 865) is an integer between `0` and `10` that represents the *replica state* (page 480) of the current member.

replSetGetStatus.**members**

The `members` field holds an array that contains a document for every member in the replica set.

replSetGetStatus.members.**name**

The `name` field holds the name of the server.

`replSetGetStatus.members.self`
The `self` field is only included in the document for the current `mongod` instance in the members array. It's value is `true`.

`replSetGetStatus.members.errmsg`
This field contains the most recent error or status message received from the member. This field may be empty (e.g. `""`) in some cases.

`replSetGetStatus.members.health`
The `health` value is only present for the other members of the replica set (i.e. not the member that returns `rs.status` (page 1017).) This field conveys if the member is up (i.e. `1`) or down (i.e. `0`.)

`replSetGetStatus.members.state`
The value of `state` (page 866) is an array of documents, each containing an integer between `0` and `10` that represents the *replica state* (page 480) of the corresponding member.

`replSetGetStatus.members.stateStr`
A string that describes `state` (page 866).

`replSetGetStatus.members.uptime`
The `uptime` (page 866) 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()` (page 1017) data.

`replSetGetStatus.members.optime`
A document that contains information regarding the last operation from the operation log that this member has applied.

`replSetGetStatus.members.optime.t`
A 32-bit timestamp of the last operation applied to this member of the replica set from the *oplog*.

`replSetGetStatus.members.optime.i`
An incremented field, which reflects the number of operations in since the last time stamp. This value only increases if there is more than one operation per second.

`replSetGetStatus.members.optimeDate`
An *ISODate* formatted date string that reflects the last entry from the *oplog* that this member applied. If this differs significantly from `lastHeartbeat` (page 866) this member is either experiencing "replication lag" *or* there have not been any new operations since the last update. Compare `members.optimeDate` between all of the members of the set.

`replSetGetStatus.members.lastHeartbeat`
The `lastHeartbeat` value provides an *ISODate* formatted date of the last heartbeat received from this member. Compare this value to the value of the `date` (page 865) field to track latency between these members.

This value does not appear for the member that returns the `rs.status()` (page 1017) data.

`replSetGetStatus.members.pingMS`
The `pingMS` represents the number of milliseconds (ms) that a round-trip packet takes to travel between the remote member and the local instance.

This value does not appear for the member that returns the `rs.status()` (page 1017) data.

`replSetGetStatus.syncingTo`
The `syncingTo` field is only present on the output of `rs.status()` (page 1017) on *secondary* and recovering members, and holds the hostname of the member from which this instance is syncing.

replSetInitiate

replSetInitiate

The replSetInitiate (page 866) 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() (page 1016) 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 473)," "*Replica Set Tutorials* (page 415)," and "*Replica Set Reconfiguration* (page 477)."

replSetMaintenance

replSetMaintenance

The replSetMaintenance (page 867) 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 (page 867) 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.

Important: On secondaries, the `compact` (page 890) command forces the secondary to enter `RECOVERING` (page 482) state. This prevents clients from reading during compaction. Once the operation finishes, the secondary returns to `SECONDARY` (page 481) state.

See *Replica Set Member States* (page 480) for more information about replica set member states. Refer to the "partial script for automating step down and compaction" for an example of this procedure.

replSetReconfig

`replSetReconfig`

The `replSetReconfig` (page 868) 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()` (page 1016) method.

Be aware of the following `replSetReconfig` (page 868) 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` (page 868) 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` (page 868) 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` (page 868) obtains a special mutually exclusive lock to prevent more than one `replSetReconfig` (page 868) operation from occurring at the same time.

replSetStepDown

Description

`replSetStepDown`

Forces the *primary* of the replica set to become a *secondary*. This initiates an *election for primary* (page 389).

field number replSetStepDown A number of seconds for the member to avoid election to primary.
If you do not specify a value for `<seconds>`, `replSetStepDown` (page 868) will attempt to avoid reelection to primary for 60 seconds.

field Boolean force New in version 2.0: Forces the *primary* to step down even if there are no secondary members within 10 seconds of the primary's latest optime.

> **Warning:** `replSetStepDown` (page 868) forces all clients currently connected to the database to disconnect. This helps 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 868) will not succeed to prevent long running elections.

Example The following example specifies that the former primary avoids reelection to primary for 120 seconds:

```
db.runCommand( { replSetStepDown: 120 } )
```

replSetSyncFrom

Description

replSetSyncFrom

New in version 2.2.

Explicitly configures which host the current `mongod` (page 1049) pulls *oplog* entries from. This operation is useful for testing different patterns and in situations where a set member is not replicating from the desired host.

The `replSetSyncFrom` (page 869) command has the following form:

```
{ replSetSyncFrom: "hostname<:port>" }
```

The `replSetSyncFrom` (page 869) command has the following field:

field string replSetSyncFrom The name and port number of the replica set member that this member should replicate from. Use the `[hostname]:[port]` form.

The Target Member

The member to replicate from must be a valid source for data in the set. The member cannot be:

- The same as the `mongod` (page 1049) on which you run `replSetSyncFrom` (page 869). In other words, a member cannot replicate from itself.
- An arbiter, because arbiters do not hold data.
- A member that does not build indexes.
- An unreachable member.
- A `mongod` (page 1049) instance that is not a member of the same replica set.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` (page 1049) will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` (page 869) during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Run from the mongo Shell

To run the command in the mongo (page 1066) shell, use the following invocation:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } )
```

You may also use the `rs.syncFrom()` (page 1018) helper in the mongo (page 1066) shell in an operation with the following form:

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

Note: `replSetSyncFrom` (page 869) and `rs.syncFrom()` (page 1018) provide a temporary override of default behavior. If:

- the mongod (page 1049) instance restarts,
- the connection to the sync target closes, or
- Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set;

then, the mongod (page 1049) instance will revert to the default sync logic and target.

resync

resync

The `resync` (page 870) command forces an out-of-date slave mongod (page 1049) instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does not apply to replica sets.

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

applyOps

Definition

applyOps

Applies specified *oplog* entries to a mongod (page 1049) instance. The `applyOps` (page 870) command is primarily an internal command to support *sharded clusters*.

The `applyOps` (page 870) command takes a document with the following fields:

field array applyOps The oplog entries to apply.

field array preCondition An array of documents that contain the conditions that must be true in order to apply the oplog entry. Each document contains a set of conditions, as described in the next table.

The `preCondition` array takes one or more documents with the following fields:

field string ns A *namespace*. If you use this field, `applyOps` (page 870) applies oplog entries only for the *collection* described by this namespace.

param string q Specifies the *query* that produces the results specified in the `res` field.

param string res The results of the query in the `q` field that must match to apply the oplog entry.

Example

The `applyOps` (page 870) command has the following prototype form:

```
db.runCommand( { applyOps: [ <operations> ], preCondition: [ { ns: <namespace>, q: <query>, res:
```

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

isMaster

Definition

isMaster

`isMaster` (page 871) returns a document that describes the role of the `mongod` (page 1049) instance.

If the instance is a member of a replica set, then `isMaster` (page 871) returns a subset of the replica set configuration and status including whether or not the instance is the *primary* of the replica set.

When sent to a `mongod` (page 1049) instance that is not a member of a replica set, `isMaster` (page 871) returns a subset of this information.

MongoDB *drivers* and *clients* use `isMaster` (page 871) to determine the state of the replica set members and to discover additional members of a *replica set*.

The `db.isMaster()` (page 1008) method in the `mongo` (page 1066) shell provides a wrapper around `isMaster` (page 871).

The command takes the following form:

```
{ isMaster: 1 }
```

See also:

`db.isMaster()` (page 1008)

Output

All Instances The following `isMaster` (page 871) fields are common across all roles:

`isMaster.`**`ismaster`**

A boolean value that reports when this node is writable. If `true`, then this instance is a *primary* in a *replica set*, or a *master* in a master-slave configuration, or a `mongos` (page 1061) instance, or a standalone `mongod` (page 1049).

This field will be `false` if the instance is a *secondary* member of a replica set or if the member is an *arbiter* of a replica set.

`isMaster.`**`maxBsonObjectSize`**

The maximum permitted size of a *BSON* object in bytes for this `mongod` (page 1049) process. If not provided, clients should assume a max size of "`4 * 1024 * 1024`".

`isMaster.`**`maxMessageSizeBytes`**

New in version 2.4.

The maximum permitted size of a *BSON* wire protocol message. The default value is `48000000` bytes.

`isMaster.`**`localTime`**

New in version 2.2.

Returns the local server time in UTC. This value is an *ISO date*.

Sharded Instances `mongos` (page 1061) instances add the following field to the `isMaster` (page 871) response document:

`isMaster.`**`msg`**

Contains the value `isdbgrid` when `isMaster` (page 871) returns from a `mongos` (page 1061) instance.

Replica Sets `isMaster` (page 871) contains these fields when returned by a member of a replica set:

`isMaster.`**`setName`**

The name of the current :replica set.

`isMaster.`**`secondary`**

A boolean value that, when `true`, indicates if the `mongod` (page 1049) is a *secondary* member of a *replica set*.

`isMaster.`**`hosts`**

An array of strings in the format of `"[hostname]:[port]"` that lists all members of the *replica set* that are neither *hidden*, *passive*, nor *arbiters*.

Drivers use this array and the `isMaster.passives` (page 872) to determine which members to read from.

`isMaster.`**`passives`**

An array of strings in the format of `"[hostname]:[port]"` listing all members of the *replica set* which have a `priority` (page 475) of `0`.

This field only appears if there is at least one member with a `priority` (page 475) of `0`.

Drivers use this array and the `isMaster.hosts` (page 872) to determine which members to read from.

`isMaster.`**`arbiters`**

An array of strings in the format of `"[hostname]:[port]"` listing all members of the *replica set* that are *arbiters*.

This field only appears if there is at least one arbiter in the replica set.

`isMaster.`**`primary`**

A string in the format of `"[hostname]:[port]"` listing the current *primary* member of the replica set.

`isMaster.`**`arbiterOnly`**

A boolean value that , when `true`, indicates that the current instance is an *arbiter*. The `arbiterOnly` (page 872) field is only present, if the instance is an arbiter.

`isMaster.`**`passive`**

A boolean value that, when `true`, indicates that the current instance is *hidden*. The `passive` (page 872) field is only present for hidden members.

`isMaster.`**`hidden`**

A boolean value that, when `true`, indicates that the current instance is *hidden*. The `hidden` (page 872) field is only present for hidden members.

`isMaster.`**`tags`**

A document that lists any tags assigned to this member. This field is only present if there are tags assigned to the member. See *Configure Replica Set Tag Sets* (page 444) for more information.

`isMaster.`**`me`**

The `[hostname]:[port]` of the member that returned `isMaster` (page 871).

getoptime

`getoptime`

`getoptime` (page 872) is an internal command.

See also:

"*Replication* (page 367)" for more information regarding replication.

Sharding Commands

Sharding Commands

Name	Description
`flushRouterConfig` (page 873)	Forces an update to the cluster metadata cached by a `mongos` (page 1061).
`addShard` (page 874)	Adds a *shard* to a *sharded cluster*.
`checkShardingIndex` (page 875)	Internal command that validates index on shard key.
`enableSharding` (page 875)	Enables sharding on a specific database.
`listShards` (page 875)	Returns a list of configured shards.
`removeShard` (page 875)	Starts the process of removing a shard from a sharded cluster.
`getShardMap` (page 875)	Internal command that reports on the state of a sharded cluster.
`getShardVersion` (page 876)	Internal command that returns the *config server* version.
`setShardVersion` (page 876)	Internal command to sets the *config server* version.
`shardCollection` (page 876)	Enables the sharding functionality for a collection, allowing the collection to be sharded.
`shardingState` (page 877)	Reports whether the `mongod` (page 1049) is a member of a sharded cluster.
`unsetSharding` (page 877)	Internal command that affects connections between instances in a MongoDB deployment.
`split` (page 878)	Creates a new *chunk*.
`splitChunk` (page 879)	Internal command to split chunk. Instead use the methods `sh.splitFind()` (page 1028) and `sh.splitAt()` (page 1028).
`splitVector` (page 880)	Internal command that determines split points.
`medianKey` (page 880)	Deprecated internal command. See `splitVector` (page 880).
`moveChunk` (page 880)	Internal command that migrates chunks between shards.
`movePrimary` (page 881)	Reassigns the *primary shard* when removing a shard from a sharded cluster.
`isdbgrid` (page 881)	Verifies that a process is a `mongos` (page 1061).

flushRouterConfig

`flushRouterConfig`

`flushRouterConfig` (page 873) clears the current cluster information cached by a `mongos` (page 1061) instance and reloads all *sharded cluster* metadata from the *config database*.

This forces an update when the configuration database holds data that is newer than the data cached in the `mongos` (page 1061) process.

> **Warning:** Do not modify the config data, except as explicitly documented. A config database cannot typically tolerate manual manipulation.

`flushRouterConfig` (page 873) is an administrative command that is only available for `mongos` (page 1061) instances.

New in version 1.8.2.

addShard

Definition

addShard

Adds either a database instance or a *replica set* to a *sharded cluster*. The optimal configuration is to deploy shards across replica sets.

Run `addShard` (page 874) when connected to a `mongos` (page 1061) instance. The command takes the following form when adding a single database instance as a shard:

```
{ addShard: "<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

When adding a replica set as a shard, use the following form:

```
{ addShard: "<replica_set>/<hostname><:port>", maxSize: <size>, name: "<shard_name>" }
```

The command contains the following fields:

field string addShard The hostname and port of the `mongod` (page 1049) instance to be added as a shard. To add a replica set as a shard, specify the name of the replica set and the hostname and port of a member of the replica set.

field integer maxSize The maximum size in megabytes of the shard. If you set `maxSize` to `0`, MongoDB does not limit the size of the shard.

field string name A name for the shard. If this is not specified, MongoDB automatically provides a unique name.

The `addShard` (page 874) command stores shard configuration information in the *config database*.

Specify a `maxSize` when you have machines with different disk capacities, or if you want to limit the amount of data on some shards. The `maxSize` constraint prevents the *balancer* from migrating chunks to the shard when the value of `mem.mapped` (page 924) exceeds the value of `maxSize`.

Examples The following command adds the database instance running on port``27027`` on the host `mongodb0.example.net` as a shard:

```
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

> **Warning:** Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The following command adds a replica set as a shard:

```
db.runCommand( { addShard: "repl0/mongodb3.example.net:27327"} )
```

You may specify all members in the replica set. All additional hostnames must be members of the same replica set.

checkShardingIndex

checkShardingIndex

checkShardingIndex (page 875) is an internal command that supports the sharding functionality.

enableSharding

enableSharding

The enableSharding (page 875) command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: "<database name>" }
```

Once you've enabled sharding in a database, you can use the shardCollection (page 876) command to begin the process of distributing data among the shards.

listShards

listShards

Use the listShards (page 875) command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

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 migrate chunks from the shard specified by [shardName]. This process happens slowly to avoid placing undue load on the overall cluster.

The command returns immediately, with the following message:

```
{ msg : "draining started successfully" , state: "started" , shard: "shardName" , ok : 1 }
```

If you run the command again, you'll see the following progress output:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 23 , dbs: 1 }, ok: 1 }
```

The remaining *document* specifies how many chunks and databases remain on the shard. Use db.printShardingStatus() (page 1010) to list the databases that you must move from the shard.

Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one of the cluster's databases, then you must manually move the database to a new shard. This can be only after the shard is empty. See the movePrimary (page 881) command for details.

After removing all chunks and databases from the shard, you may issue the command again, to return:

```
{ msg: "remove shard completed successfully", state: "completed", host: "shardName", ok : 1 }
```

getShardMap

getShardMap

getShardMap (page 875) is an internal command that supports the sharding functionality.

getShardVersion

getShardVersion

`getShardVersion` (page 876) is an internal command that supports sharding functionality.

setShardVersion

setShardVersion

`setShardVersion` (page 876) is an internal command that supports sharding functionality.

shardCollection

Definition

shardCollection

Enables a collection for sharding and allows MongoDB to begin distributing data among shards. You must run `enableSharding` (page 875) on a database before running the `shardCollection` (page 876) command. `shardCollection` (page 876) has the following form:

```
{ shardCollection: "<database>.<collection>", key: <shardkey> }
```

`shardCollection` (page 876) has the following fields:

field string shardCollection The *namespace* of the collection to shard in the form `<database>.<collection>`.

field document key The *index specification document* (page 66) to use as the shard key. The index must exist prior to the `shardCollection` (page 876) command, unless the collection is empty. If the collection is empty, in which case MongoDB creates the index prior to sharding the collection. New in version 2.4: The key may be in the form `{ field : "hashed" }`, which will use the specified field as a hashed shard key.

field Boolean unique When `true`, the `unique` option ensures that the underlying index enforces a unique constraint. Hashed shard keys do not support unique constraints.

field integer numInitialChunks To support *hashed sharding* (page 502) added in MongoDB 2.4, `numInitialChunks` specifies the number of chunks to create when sharding an collection with a hashed shard key. MongoDB will then create and balance chunks across the cluster. The `numInitialChunks` must be less than `8192`.

Shard Keys Choosing the best shard key to effectively distribute load among your shards requires some planning. Review *Shard Keys* (page 502) regarding choosing a shard key.

Hashed Shard Keys New in version 2.4.

Hashed shard keys (page 502) use a hashed index of a single field as the shard key.

> **Warning:** MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 876). Additionally, after `shardCollection` (page 876), you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

Sharding (page 485), *Sharding Concepts* (page 495), and *Deploy a Sharded Cluster* (page 516).

Example The following operation enables sharding for the `people` collection in the `records` database and uses the `zipcode` field as the *shard key* (page 502):

```
db.runCommand( { shardCollection: "records.people", key: { zipcode: 1 } } )
```

shardingState

shardingState

`shardingState` (page 877) is an admin command that reports if `mongod` (page 1049) is a member of a *sharded cluster*. `shardingState` (page 877) has the following prototype form:

```
{ shardingState: 1 }
```

For `shardingState` (page 877) to detect that a `mongod` (page 1049) is a member of a sharded cluster, the `mongod` (page 1049) must satisfy the following conditions:

1. the `mongod` (page 1049) is a primary member of a replica set, and
2. the `mongod` (page 1049) instance is a member of a sharded cluster.

If `shardingState` (page 877) detects that a `mongod` (page 1049) is a member of a sharded cluster, `shardingState` (page 877) returns a document that resembles the following prototype:

```
{
  "enabled" : true,
  "configServer" : "<configdb-string>",
  "shardName" : "<string>",
  "shardHost" : "string:",
  "versions" : {
       "<database>.<collection>" : Timestamp(<...>),
       "<database>.<collection>" : Timestamp(<...>)
  },
  "ok" : 1
}
```

Otherwise, `shardingState` (page 877) will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from `shardingState` (page 877) when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: `mongos` (page 1061) instances do not provide the `shardingState` (page 877).

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.

unsetSharding

unsetSharding

`unsetSharding` (page 877) is an internal command that supports sharding functionality.

split

Definition

split

Splits a *chunk* in a *sharded cluster* into two chunks. In most cases, you do not need to do this. The `mongos` (page 1061) instance splits and manages chunks automatically. However, the `split` (page 878) command does allow administrators to manually create splits.

The `split` (page 878) command is an administrative command that is only available on `mongos` (page 1061) instances.

The `split` (page 878) command takes a document with the following fields:

field string split The name of the *collection* where the *chunk* exists. Specify the collection's full *namespace*, including the database name.

field document find An query statement that specifies an equality match on the shard key. The match selects the chunk that contains the specified document. You must specify only one of the following: `find`, `bounds`, or `middle`.

field array bounds New in version 2.4: The bounds of a chunk to split. `bounds` applies to chunks in collections partitioned using a *hashed shard key*. The parameter's array must consist of two documents specifying the lower and upper shard-key values of the chunk. The values must match the minimum and maximum values of an existing chunk. Specify only one of the following: `find`, `bounds`, or `middle`.

field document middle The document to use as the split point to create two chunks. `split` (page 878) requires one of the following options: `find`, `bounds`, or `middle`.

Command Formats To issue the `split` (page 878) command, connect to the `admin` database of a `mongos` (page 1061) instance. Pass to the `db.runCommand()` (page 1012) method a document of one of the following forms:

```
{ split: <database>.<collection>, find: <document> }
```

Or:

```
{ split: <database>.<collection>, middle: <document> }
```

Or:

```
{ split: <database>.<collection>, bounds: [ <lower>, <upper> ] }
```

To create a split for a collection that uses a *hashed shard key*, use the `bounds` parameter. Do *not* use the `middle` parameter.

> **Warning:** Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

Examples The following sections provide examples of the count command.

Split a Chunk in Half

```
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` (page 878) 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.

Define an Arbitrary Split Point 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 a Chunk Using Values of a Hashed Shard Key To split a specific chunk using the minimum and maximum values of the *hashed shard key* of that chunk, use the following:

```
db.runCommand( { split: "test.people" ,
                 bounds : [ NumberLong("-5838464104018346494"),
                            NumberLong("-5557153028469814163")] } )
```

Metadata Lock Error If another process, such as a balancer process, changes meta data while `split` (page 878) is running, you may see this error. You may retry the `split` (page 878) operation without side effects.

```
errmsg: "The collection's metadata lock is already taken."
```

See also:

`moveChunk` (page 880), `sh.moveChunk()` (page 1026), `sh.splitAt()` (page 1028), and `sh.splitFind()` (page 1028).

splitChunk

Definition

`splitChunk`

An internal administrative command. To split chunks, use the `sh.splitFind()` (page 1028) and `sh.splitAt()` (page 1028) functions in the `mongo` (page 1066) shell.

> **Warning:** Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

See also:

`moveChunk` (page 880) and `sh.moveChunk()` (page 1026).

The `splitChunk` (page 879) command takes a document with the following fields:

field string ns The complete *namespace* of the *chunk* to split.

field document keyPattern The *shard key*.

field document min The lower bound of the shard key for the chunk to split.

field document max The upper bound of the shard key for the chunk to split.

field string from The *shard* that owns the chunk to split.

field document splitKeys The split point for the chunk.

field document shardId The shard.

splitVector

splitVector

Is an internal command that supports meta-data operations in sharded clusters.

medianKey

medianKey

`medianKey` (page 880) is an internal command.

moveChunk

Definition

moveChunk

Internal administrative command. Moves *chunks* between *shards*. Issue the `moveChunk` (page 880) command via a `mongos` (page 1061) instance while using the *admin database*. Use the following form:

```
db.runCommand( { moveChunk : <namespace> ,
                 find|bounds : <query|array> ,
                 to : <string>,
                 _secondaryThrottle : <Boolean> } )
```

The `moveChunk` (page 880) command has the following fields:

field string moveChunk The *namespace* of the *collection* where the *chunk* exists. Specify the collection's full namespace, including the database name.

field document find An equality match on the shard key that specifies the shard-key value of the chunk to move. Specify either the `bounds` field or the `find` field but not both.

field array bounds The bounds of a specific chunk to move. The array must consist of two documents that specify the lower and upper shard key values of a chunk to move. Specify either the `bounds` field or the `find` field but not both. Use `bounds` to move chunks in collections partitioned using a *hashed shard key*.

field string to The name of the destination shard for the chunk.

field Boolean _secondaryThrottle Defaults to `true`. When `true`, the balancer waits for replication to *secondaries* when it copies and deletes data during chunk migrations. For details, see *Require Replication before Chunk Migration (Secondary Throttle)* (page 541).

The *chunk migration* (page 511) section describes how chunks move between shards on MongoDB.

See also:

`split` (page 878), `sh.moveChunk()` (page 1026), `sh.splitAt()` (page 1028), and `sh.splitFind()` (page 1028).

Return Messages `moveChunk` (page 880) returns the following message if another metadata operation is in progress `chunks` (page 557) collection:

```
errmsg: "The collection's metadata lock is already taken."
```

If another process, such as a balancer process, changes meta data while moveChunk (page 880) is running, you may see this error. You may retry the moveChunk (page 880) operation without side effects.

Note: Only use the moveChunk (page 880) in special circumstances such as preparing your *sharded cluster* for an initial ingestion of data, or a large bulk import operation. In most cases allow the balancer to create and balance chunks in sharded clusters. See *Create Chunks (Pre-Splitting)* (page 537) 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 (page 881) is an administrative command that is only available for mongos (page 1061) instances. Only use movePrimary (page 881) when removing a shard from a sharded cluster.

Important: Only use movePrimary (page 881) when:

- the database does not contain any collections with data, *or*
- you have drained all sharded collections using the removeShard (page 875) command.

See *Remove Shards from an Existing Sharded Cluster* (page 544) for a complete procedure.

movePrimary (page 881) 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 (page 875) command.

isdbgrid

isdbgrid

This command verifies that a process is a mongos (page 1061).

If you issue the isdbgrid (page 881) command when connected to a mongos (page 1061), 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 (page 881) command when connected to a mongod (page 1049), MongoDB returns an error document. The isdbgrid (page 881) command is not available to mongod (page 1049). The error document, however, also includes a line that reads `"isdbgrid" : 1`, just as in the document returned for a mongos (page 1061). 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 (page 871) command to determine connection to a mongos (page 1061). When connected to a mongos (page 1061), the isMaster (page 871) command returns a document that contains the string `isdbgrid` in the `msg` field.

See also:

Sharding (page 485) for more information about MongoDB's sharding functionality.

Instance Administration Commands

Administration Commands

Name	Description
`renameCollection` (page 882)	Changes the name of an existing collection.
`copydb` (page 883)	Copies a database from a remote host to the current host.
`dropDatabase` (page 884)	Removes the current database.
`drop` (page 884)	Removes the specified collection from the database.
`create` (page 885)	Creates a collection and sets collection parameters.
`clone` (page 885)	Copies a database from a remote host to the current host.
`cloneCollection` (page 886)	Copies a collection from a remote host to the current host.
`cloneCollectionAsCapped` (page 887)	Copies a non-capped collection as a new *capped collection*.
`closeAllDatabases` (page 887)	Internal command that invalidates all cursors and closes open database files.
`convertToCapped` (page 887)	Converts a non-capped collection to a capped collection.
`filemd5` (page 888)	Returns the *md5* hash for files stored using *GridFS*.
`dropIndexes` (page 888)	Removes indexes from a collection.
`fsync` (page 888)	Flushes pending writes to the storage layer and locks the database to allow backups.
`clean` (page 890)	Internal namespace administration command.
`connPoolSync` (page 890)	Internal command to flush connection pool.
`compact` (page 890)	Defragments a collection and rebuilds the indexes.
`collMod` (page 892)	Add flags to collection to modify the behavior of MongoDB.
`reIndex` (page 894)	Rebuilds all indexes on a collection.
`setParameter` (page 894)	Modifies configuration options.
`getParameter` (page 895)	Retrieves configuration options.
`repairDatabase` (page 895)	Repairs any errors and inconsistencies with the data storage.
`touch` (page 896)	Loads documents and indexes from data storage to memory.
`shutdown` (page 896)	Shuts down the `mongod` (page 1049) or `mongos` (page 1061) process.
`logRotate` (page 897)	Rotates the MongoDB logs to prevent a single file from taking too much space.

renameCollection

Definition

renameCollection

Changes the name of an existing collection. Specify collections to `renameCollection` (page 882) in the form of a complete *namespace*, which includes the database name. Issue the `renameCollection` (page 882) command against the *admin database*. The command takes the following form:

```
{ renameCollection: "<source_namespace>", to: "<target_namespace>", dropTarget: <true|false> }
```

The command contains the following fields:

field string renameCollection The *namespace* of the collection to rename. The namespace is a combination of the database name and the name of the collection.

field string to The new namespace of the collection. If the new namespace specifies a different database, the `renameCollection` (page 882) command copies the collection to the new database and drops the source collection.

field boolean dropTarget If `true`, `mongod` (page 1049) will drop the `target` of `renameCollection` (page 882) prior to renaming the collection.

`renameCollection` (page 882) is suitable for production environments; *however*:

- `renameCollection` (page 882) blocks all database activity for the duration of the operation.
- `renameCollection` (page 882) is **not** compatible with sharded collections.

> **Warning:** `renameCollection` (page 882) fails if `target` is the name of an existing collection *and* you do not specify `dropTarget: true`.
> If the `renameCollection` (page 882) operation does not complete the `target` collection and indexes will not be usable and will require manual intervention to clean up.

Exceptions

exception 10026 Raised if the `source` namespace does not exist.

exception 10027 Raised if the `target` namespace exists and `dropTarget` is either `false` or unspecified.

exception 15967 Raised if the `target` namespace is an invalid collection name.

Shell Helper The shell helper `db.collection.renameCollection()` (page 971) 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` (page 882) with sharded collections.

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

copydb

copydb

The `copydb` (page 883) 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>,
  nonce: <nonce>,
  key: <key> }
```

All of the following arguments are optional:

- slaveOk
- username

- 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 destination, or the `todb` server.

Be aware of the following behaviors:

- `copydb` (page 883) 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` (page 883) 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` (page 883) on the **destination server**.
- The destination server is not locked for the duration of the `copydb` (page 883) operation. This means that `copydb` (page 883) will occasionally yield to allow other operations to complete.
- If the remote server has authentication enabled, then you must include a username, 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(<remote_db_name>, <local_db_name>, <from_host_name>, <username>, <password>)
```

dropDatabase

`dropDatabase`

The `dropDatabase` (page 884) command drops a database, deleting the associated data files. `dropDatabase` (page 884) 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` (page 1066) 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.

drop

`drop`

The `drop` (page 884) command removes an entire collection from a database. The command has following syntax:

```
{ drop: <collection_name> }
```

The `mongo` (page 1066) 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.

create

Definition

create

Explicitly creates a collection. `create` (page 885) has the following form:

```
{ create: <collection_name>,
  capped: <true|false>,
  autoIndexId: <true|false>,
  size: <max_size>,
  max: <max_documents>
}
```

`create` (page 885) has the following fields:

field string create The name of the new collection.

field Boolean capped To create a *capped collection*. specify `true`. If you specify `true`, you must also set a maximum size in the `size` field.

field Boolean autoIndexID Specify `false` to disable the automatic creation of an index on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`.

field integer size The maximum size for the capped collection. Once a capped collection reaches its maximum size, MongoDB overwrites older old documents with new documents. The `size` field is required for capped collections.

field integer max The maximum number of documents to keep in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches its maximum size before it reaches the maximum number of documents, MongoDB removes old documents. If you use this limit, ensure that the `size` limit is sufficient to contain the documents limit.

For more information on the `autoIndexId` field in versions before 2.2, see *_id Fields and Indexes on Capped Collections* (page 1194).

The `db.createCollection()` (page 997) method wraps the `create` (page 885) command.

> **Note:** The `create` (page 885) 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.

Example To create a *capped collection* limited to 64 kilobytes, issue the command in the following form:

```
db.runCommand( { create: "collection", capped: true, size: 64 * 1024 } )
```

clone

clone

The `clone` (page 885) command clone a database from a remote MongoDB instance to the current host. `clone` (page 885) 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` (page 885) can run against a *slave* or a non-*primary* member of a *replica set*.
- `clone` (page 885) 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` (page 885) on the **destination server**.
- The destination server is not locked for the duration of the `clone` (page 885) operation. This means that `clone` (page 885) will occasionally yield to allow other operations to complete.

See `copydb` (page 883) for similar functionality.

> **Warning:** This command obtains an intermittent *write lock* on the destination server, that can block other operations until it completes.

cloneCollection

Definition

`cloneCollection`

Copies a collection from a remote `mongod` (page 1049) instance to the current `mongod` (page 1049) instance. `cloneCollection` (page 886) creates a collection in a database with the same name as the remote collection's database. `cloneCollection` (page 886) takes the following form:

```
{ cloneCollection: "<collection>", from: "<hostname>", query: { <query> }, copyIndexes: <true|fa
```

Important: You cannot clone a collection through a `mongos` (page 1061) but must connect directly to the `mongod` (page 1049) instance.

`cloneCollection` (page 886) has the following fields:

field string cloneCollection The name of the collection to clone.

field string from Specify a resolvable hostname and optional port number of the remote server where the specified collection resides.

field document query A query that filters the documents in the remote collection that `cloneCollection` (page 886) will copy to the current database.

field boolean copyIndexes If set to `false` the indexes on the originating server are not copied with the documents in the collection. This is set to `true` by default.

Example

```
{ cloneCollection: "users.profiles", from: "mongodb.example.net:27017", query: { active: true }, copy
```

This operation copies the `profiles` collection from the `users` database on the server at `mongodb.example.net`. The operation only copies documents that satisfy the query `{ active: true }` and does not copy indexes. `cloneCollection` (page 886) copies indexes by default, but you can disable this behavior by setting `{ copyIndexes: false }`. The `query` and `copyIndexes` arguments are optional.

If, in the above example, the `profiles` collection exists in the `users` database, then MongoDB appends documents from the remote collection to the destination collection.

cloneCollectionAsCapped

cloneCollectionAsCapped

The cloneCollectionAsCapped (page 887) 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 (page 887) command.

During the cloning, the cloneCollectionAsCapped (page 887) 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 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

closeAllDatabases

closeAllDatabases (page 887) 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.

convertToCapped

convertToCapped

The convertToCapped (page 887) 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 (page 887) 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 (page 887) command exhibit the following behavior:

- MongoDB transverses the documents in the original collection in *natural order* and loads the documents 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 in the capped collection based on insertion order, or *first in, first out* order.
- Internally, to convert the collection, MongoDB uses the following procedure
  - cloneCollectionAsCapped (page 887) command creates the capped collection and imports the data.
  - MongoDB drops the original collection.

–renameCollection (page 882) renames the new capped collection to the name of the original collection.

Note: MongoDB does not support the convertToCapped (page 887) command in a sharded cluster.

Warning: The convertToCapped (page 887) will not recreate indexes from the original collection on the new collection, other than the index on the _id field. If you need indexes on this collection you will need to create these indexes after the conversion is complete.

See also:

create (page 885)

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

filemd5

filemd5

The filemd5 (page 888) command returns the *md5* hashes for a single file 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" }
```

dropIndexes

dropIndexes

The dropIndexes (page 888) 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.

fsync

fsync

The fsync (page 888) command forces the mongod (page 1049) process to flush all pending writes to the storage layer. mongod (page 1049) 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 1122) interval, which is 60 seconds by default.

```
{ fsync: 1 }
```

The fsync (page 888) operation is synchronous by default, to run fsync (page 888) asynchronously, use the following form:

```
{ fsync: 1, async: true }
```

The connection will return immediately. You can check the output of db.currentOp() (page 998) for the status of the fsync (page 888) operation.

The primary use of fsync (page 888) 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 (page 888) 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() (page 998). 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 (page 1066) 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() (page 1004) and db.fsyncUnlock() (page 1004) helpers in the shell.

In the mongo (page 1066) shell, you may use the db.fsyncLock() (page 1004) and db.fsyncUnlock() (page 1004) wrappers for the fsync (page 888) lock and unlock process:

```
db.fsyncLock();
db.fsyncUnlock();
```

Note: fsync (page 888) 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 *Sharded Cluster Backup Considerations* (page 134).

If your mongod (page 1049) has *journaling* enabled, consider using *another method* (page 139) to back up your database.

Note: The database cannot be locked with db.fsyncLock() (page 1004) while profiling is enabled. You must disable profiling before locking the database with db.fsyncLock() (page 1004). Disable profiling using db.setProfilingLevel() (page 1012) as follows in the mongo (page 1066) shell:

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

clean

clean

`clean` (page 890) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

connPoolSync

connPoolSync

`connPoolSync` (page 890) is an internal command.

compact

Definition

compact

New in version 2.0.

Rewrites and defragments all data in a collection, as well as all of the indexes on that collection. `compact` (page 890) has the following form:

```
{ compact: <collection name> }
```

`compact` (page 890) has the following fields:

field string compact The name of the collection.

field boolean force If `true`, `compact` (page 890) can run on the *primary* in a *replica set*. If `false`, `compact` (page 890) returns an error when run on a primary, because the command blocks all other activity. Beginning with version 2.2, `compact` (page 890) blocks activity only for the database it is compacting.

field number paddingFactor Describes the *record size* allocated for each document as a factor of the document size for all records compacted during the `compact` (page 890) operation. The `paddingFactor` does not affect the padding of subsequent record allocations after `compact` (page 890) completes. For more information, see *paddingFactor* (page 890).

field integer paddingBytes Sets the padding as an absolute number of bytes for all records compacted during the `compact` (page 890) operation. After `compact` (page 890) completes, `paddingBytes` does not affect the padding of subsequent record allocations. For more information, see *paddingBytes* (page 891).

`compact` (page 890) is similar to `repairDatabase` (page 895); however, `repairDatabase` (page 895) operates on an entire database.

paddingFactor New in version 2.2.

The `paddingFactor` field takes the following range of values:

- Default: `1.0`
- Minimum: `1.0` (no padding)
- Maximum: `4.0`

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` (page 890) 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 } )
```

`compact` (page 890) compacts existing documents but does not reset `paddingFactor` statistics for the collection. After the `compact` (page 890) MongoDB will use the existing `paddingFactor` when allocating new records for documents in this collection.

paddingBytes New in version 2.2.

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` (page 890) 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` (page 890) operation.

Behaviors The `compact` (page 890) has the behaviors described here.

Blocking In MongoDB 2.2, `compact` (page 890) blocks activities only for its database. Prior to 2.2, the command blocked all activities.

You may view the intermediate progress either by viewing the `mongod` (page 1049) log file or by running the `db.currentOp()` (page 998) in another shell instance.

Operation Termination If you terminate the operation with the `db.killOp()` (page 1009) method or restart the server before the `compact` (page 890) operation has finished:

- If you have journaling enabled, the data remains consistent and usable, regardless of the state of the `compact` (page 890) operation. You may have to manually rebuild the indexes.
- If you do not have journaling enabled and the `mongod` (page 1049) or `compact` (page 890) terminates during the operation, it is 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.

Disk Space `compact` (page 890) generally uses less disk space than `repairDatabase` (page 895) and is faster. However, the `compact` (page 890) command is still slow and blocks other database use. Only use `compact` (page 890) during scheduled maintenance periods.

`compact` (page 890) requires up to 2 gigabytes of additional disk space while running. Unlike `repairDatabase` (page 895), `compact` (page 890) does *not* free space on the file system.

To see how the storage space changes for the collection, run the `collStats` (page 900) command before and after compaction.

Size and Number of Data Files `compact` (page 890) may increase the total size and number of your 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.

Replica Sets `compact` (page 890) commands do not replicate to secondaries in a *replica set*:

- Compact each member separately.
- Ideally run `compact` (page 890) on a secondary. See option `force:true` above for information regarding compacting the primary.

  Important: On secondaries, the `compact` (page 890) command forces the secondary to enter `RECOVERING` (page 482) state. This prevents clients from reading during compaction. Once the operation finishes, the secondary returns to `SECONDARY` (page 481) state.

  See *Replica Set Member States* (page 480) for more information about replica set member states. Refer to the "partial script for automating step down and compaction" for an example of this procedure.

Sharded Clusters `compact` (page 890) is a command issued to a `mongod` (page 1049). In a sharded environment, run `compact` (page 890) on each shard separately as a maintenance operation.

Important: You cannot issue `compact` (page 890) against a `mongos` (page 1061) instance.

Capped Collections 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 collection* are not subject to fragmentation.

collMod

`collMod`

New in version 2.2.

`collMod` (page 892) makes it possible to add flags to a collection to modify the behavior of MongoDB. Flags include `usePowerOf2Sizes` (page 892) and `index` (page 893). The command takes the following prototype form:

```
db.runCommand( {"collMod" : <collection> , "<flag>" : <value> } )
```

In this command substitute `<collection>` with the name of a collection in the current database, and `<flag>` and `<value>` with the flag and value you want to set.

Use the `userFlags` (page 902) field in the in `db.collection.stats()` (page 973) output to check enabled collection flags.

usePowerOf2Sizes

The usePowerOf2Sizes (page 892) flag changes the method that MongoDB uses to allocate space on disk for documents in this collection. By setting usePowerOf2Sizes (page 892), 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 usePowerOf2Sizes (page 892) MongoDB will be able to more effectively reuse space.

Note: With usePowerOf2Sizes (page 892) MongoDB, allocates records that have power of 2 sizes, until record sizes equal 4 megabytes. For records larger than 4 megabytes with usePowerOf2Sizes (page 892) set, mongod (page 1049) will allocate records in full megabytes by rounding up to the nearest megabyte.

---

usePowerOf2Sizes (page 892) 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.

Example

To enable usePowerOf2Sizes (page 892) on the collection named `products`, use the following operation:

```
db.runCommand( {collMod: "products", usePowerOf2Sizes : true })
```

To disable usePowerOf2Sizes (page 892) on the collection `products`, use the following operation:

```
db.runCommand( { collMod: "products", usePowerOf2Sizes: false })
```

Warning: Changed in version 2.2.1: usePowerOf2Sizes (page 892) now supports documents larger than 8 megabytes. If you enable usePowerOf2Sizes (page 892) you **must** use at least version 2.2.1.

usePowerOf2Sizes (page 892) only affects subsequent allocations caused by document insertion or record relocation as a result of document growth, and *does not* affect existing allocations.

index

The index (page 893) flag changes the expiration time of a *TTL Collection* (page 599).

Specify the key and new expiration time with a document of the form:

```
{keyPattern: <index_spec>, expireAfterSeconds: <seconds> }
```

where <index_spec> is an existing index in the collection and `seconds` is the number of seconds to subtract from the current time.

Example

To update the expiration value for a collection named `sessions` indexed on a `lastAccess` field from 30 minutes to 60 minutes, use the following operation:

```
db.runCommand({collMod: "sessions",
               index: {keyPattern: {lastAccess:1},
               expireAfterSeconds: 3600}})
```

Which will return the document:

```
{ "expireAfterSeconds_old" : 1800, "expireAfterSeconds_new" : 3600, "ok" : 1 }
```

On success collMod (page 892) returns a document with fields `expireAfterSeconds_old` and `expireAfterSeconds_new` set to their respective values.

On failure, `collMod` (page 892) returns a document with `no expireAfterSeconds field to update` if there is no existing `expireAfterSeconds` field or `cannot find index { **key**: 1.0 } for ns **namespace**` if the specified `keyPattern` does not exist.

reIndex

`reIndex`

The `reIndex` (page 894) 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` (page 894) command 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.

Call `reIndex` (page 894) 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.

> **Note:** For replica sets, `reIndex` (page 894) will not propagate from the *primary* to *secondaries*. `reIndex` (page 894) will only affect a single `mongod` (page 1049) instance.

setParameter

`setParameter`

`setParameter` (page 894) is an administrative command for modifying options normally set on the command line. You must issue the `setParameter` (page 894) command against the *admin database* in the form:

```
{ setParameter: 1, <option>: <value> }
```

Replace the `<option>` with one of the supported `setParameter` (page 894) options:

- `journalCommitInterval` (page 1129)
- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `replApplyBatchSize` (page 1130)
- `replIndexPrefetch` (page 1130)
- `syncdelay` (page 1131)
- `traceExceptions` (page 1131)
- `textSearchEnabled` (page 1132)

getParameter

getParameter

getParameter (page 895) 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 (page 894) for more about these parameters.

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 (page 895) or related options like db.repairDatabase() (page 1011) in the mongo (page 1066) shell or *mongod --repair*. Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run repairDatabase (page 895). In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

The repairDatabase (page 895) 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 (page 1049) 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 (page 895) command to ensure that there are no errors in the data storage.

As a side effect, the repairDatabase (page 895) command will compact the database, as the compact (page 890) command, and also reduces the total size of the data files on disk. The repairDatabase (page 895) 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* 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() (page 1011) in the mongo (page 1066) shell.
- Run mongod (page 1049) directly from your system's shell. Make sure that mongod (page 1049) 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* or *--slave* options, as necessary.

touch

touch

New in version 2.2.

The touch (page 896) command loads data from the data storage layer into memory. touch (page 896) 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 (page 1049) will ideally be able to perform subsequent operations more efficiently. The touch (page 896) command has the following prototypical form:

```
{ touch: [collection], data: [boolean], index: [boolean] }
```

By default, data and index are false, and touch (page 896) 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 (page 1066) shell:

```
db.runCommand({ touch: "records", data: true, index: true })
```

touch (page 896) will not block read and write operations on a mongod (page 1049), and can run on *secondary* members of replica sets.

Note: Using touch (page 896) to control or tweak what a mongod (page 1049) stores in memory may displace other records data in memory and hinder performance. Use with caution in production systems.

Warning: If you run touch (page 896) on a secondary, the secondary will enter a RECOVERING state to prevent clients from sending read operations during the touch (page 896) operation. When touch (page 896) finishes the secondary will automatically return to SECONDARY state. See state (page 866) for more information on replica set member states.

shutdown

shutdown

The shutdown (page 896) command cleans up all database resources and then terminates the process. You must issue the shutdown (page 896) command against the *admin database* in the form:

```
{ shutdown: 1 }
```

Note: Run the shutdown (page 896) against the *admin database*. When using shutdown (page 896), 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 367) 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` (page 896) 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` (page 1066) shell helper syntax looks like this:

```
db.shutdownServer({timeoutSecs: 60});
```

logRotate

`logRotate`

The `logRotate` (page 897) 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` (page 897) command against the *admin database* in the form:

```
{ logRotate: 1 }
```

Note: Your `mongod` (page 1049) instance needs to be running with the `--logpath [file]` option.

You may also rotate the logs by sending a `SIGUSR1` signal to the `mongod` (page 1049) process. If your `mongod` (page 1049) has a process ID of 2200, here's how to send the signal on Linux:

```
kill -SIGUSR1 2200
```

`logRotate` (page 897) 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` (page 897) creates a new log file with the same name as originally specified by the `logpath` (page 1116) setting to `mongod` (page 1049) or `mongos` (page 1061).

Note: New in version 2.0.3: The `logRotate` (page 897) command is available to `mongod` (page 1049) instances running on Windows systems with MongoDB release 2.0.3 and higher.

Diagnostic Commands

Diagnostic Commands

Name	Description
listDatabases (page 898)	Returns a document that lists all databases and returns basic database statistics.
dbHash (page 899)	Internal command to support sharding.
driverOIDTest (page 899)	Internal command that converts an ObjectID to a string to support tests.
listCommands (page 899)	Lists all database commands provided by the current mongod (page 1049) instance.
availableQueryOptions (page 899)	Internal command that reports on the capabilities of the current MongoDB instance.
buildInfo (page 899)	Displays statistics about the MongoDB build.
collStats (page 900)	Reports storage utilization statics for a specified collection.
connPoolStats (page 902)	Reports statistics on the outgoing connections from this MongoDB instance to other MongoDB instances in the deployment.
dbStats (page 904)	Reports storage utilization statistics for the specified database.
cursorInfo (page 906)	Reports statistics on active cursors.
dataSize (page 906)	Returns the data size for a range of data. For internal use.
diagLogging (page 906)	Provides a diagnostic logging. For internal use.
getCmdLineOpts (page 906)	Returns a document with the run-time arguments to the MongoDB instance and their parsed options.
netstat (page 907)	Internal command that reports on intra-deployment connectivity. Only available for mongos (page 1061) instances.
ping (page 907)	Internal command that tests intra-deployment connectivity.
profile (page 907)	Interface for the *database profiler* (page 1134).
validate (page 908)	Internal command that scans for a collection's data and indexes for correctness.
top (page 911)	Returns raw usage statistics for each database in the mongod (page 1049) instance.
indexStats (page 911)	Experimental command that collects and aggregates statistics on all indexes.
whatsmyuri (page 916)	Internal command that returns information on the current client.
getLog (page 916)	Returns recent log messages.
hostInfo (page 917)	Returns data that reflects the underlying host system.
serverStatus (page 919)	Returns a collection metrics on instance-wide resource utilization and status.
features (page 936)	Reports on features available in the current MongoDB instance.
isSelf	Internal command to support testing.

listDatabases

listDatabases

The listDatabases (page 898) command provides a list of existing databases along with basic statistics about them:

```
{ listDatabases: 1 }
```

The value (e.g. 1) does not affect the output of the command. listDatabases (page 898) 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.

dbHash

dbHash

dbHash (page 899) is an internal command.

driverOIDTest

driverOIDTest

driverOIDTest (page 899) is an internal command.

listCommands

listCommands

The listCommands (page 899) command generates a list of all database commands implemented for the current mongod (page 1049) instance.

availableQueryOptions

availableQueryOptions

availableQueryOptions (page 899) is an internal command that is only available on mongos (page 1061) instances.

buildInfo

buildInfo

The buildInfo (page 899) command is an administrative command which returns a build summary for the current mongod (page 1049). buildInfo (page 899) has the following prototype form:

```
{ buildInfo: 1 }
```

In the mongo (page 1066) shell, call buildInfo (page 899) in the following form:

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

Example

The output document of buildInfo (page 899) has the following form:

```
{
  "version" : "<string>",
  "gitVersion" : "<string>",
  "sysInfo" : "<string>",
  "loaderFlags" : "<string>",
  "compilerFlags" : "<string>",
  "allocator" : "<string>",
  "versionArray" : [ <num>, <num>, <...> ],
  "javascriptEngine" : "<string>",
  "bits" : <num>,
  "debug" : <boolean>,
  "maxBsonObjectSize" : <num>,
  "ok" : <num>
}
```

Consider the following documentation of the output of buildInfo (page 899):

buildInfo

The document returned by the buildInfo (page 899) command.

buildInfo.**gitVersion**

The commit identifier that identifies the state of the code used to build the mongod (page 1049).

buildInfo.**sysInfo**
A string that holds information about the operating system, hostname, kernel, date, and Boost version used to compile the `mongod` (page 1049).

buildInfo.**loaderFlags**
The flags passed to the loader that loads the `mongod` (page 1049).

buildInfo.**compilerFlags**
The flags passed to the compiler that builds the `mongod` (page 1049) binary.

buildInfo.**allocator**
Changed in version 2.2.

The memory allocator that `mongod` (page 1049) uses. By default this is `tcmalloc` after version 2.2, and `system` before 2.2.

buildInfo.**versionArray**
An array that conveys version information about the `mongod` (page 1049) instance. See `version` for a more readable version of this string.

buildInfo.**javascriptEngine**
Changed in version 2.4.

A string that reports the JavaScript engine used in the `mongod` (page 1049) instance. By default, this is `V8` after version 2.4, and `SpiderMonkey` before 2.4.

buildInfo.**bits**
A number that reflects the target processor architecture of the `mongod` (page 1049) binary.

buildInfo.**debug**
A boolean. `true` when built with debugging options.

buildInfo.**maxBsonObjectSize**
A number that reports the `Maximum BSON Document Size` (page 1139).

collStats

Definition

collStats
The `collStats` (page 900) command returns a variety of storage statistics for a given collection. Use the following syntax:

```
{ collStats: "collection" , scale : 1024 }
```

Specify the `collection` you want statistics for, 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()` (page 973) helper in the `mongo` (page 1066) 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.

Output

`collStats.`**`ns`**

The namespace of the current collection, which follows the format `[database].[collection]`.

`collStats.`**`count`**

The number of objects or documents in this collection.

`collStats.`**`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 902) field reports.

The `scale` argument affects this value.

`collStats.`**`avgObjSize`**

The average size of an object in the collection. The `scale` argument affects this value.

`collStats.`**`storageSize`**

The total amount of storage allocated to this collection for *document* storage. The `scale` argument affects this value. The `storageSize` (page 901) does not decrease as you remove or shrink documents.

`collStats.`**`numExtents`**

The total number of contiguously allocated data file regions.

`collStats.`**`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` (page 1049) may not have an `_id` index.

`collStats.`**`lastExtentSize`**

The size of the last extent allocated. The `scale` argument affects this value.

`collStats.`**`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` (page 1049) automatically calculates this padding factor

`collStats.`**`flags`**

Changed in version 2.2: Removed in version 2.2 and replaced with the `userFlags` (page 902) and `systemFlags` (page 901) 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.

`collStats.`**`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.

collStats.**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 (page 892). If usePowerOf2Sizes (page 892) is enabled, userFlags (page 902) will be set to 1, otherwise userFlags (page 902) will be 0.

See the collMod (page 892) command for more information on setting user flags and usePowerOf2Sizes (page 892).

collStats.**totalIndexSize**
The total size of all indexes. The scale argument affects this value.

collStats.**indexSizes**
This field specifies the key and size of every existing index on the collection. The scale argument affects this value.

Example The following is an example of db.collection.stats() (page 973) and collStats (page 900) output:

```
{
     "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
}
```

connPoolStats

Definition

connPoolStats

Note: connPoolStats (page 902) only returns meaningful results for mongos (page 1061) instances and for mongod (page 1049) instances in sharded clusters.

The command connPoolStats (page 902) 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.

Note: `connPoolStats` (page 902) only returns meaningful results for `mongos` (page 1061) instances and for `mongod` (page 1049) instances in sharded clusters.

Output

`connPoolStats.hosts`

The sub-documents of the `hosts` (page 903) *document* report connections between the `mongos` (page 1061) or `mongod` (page 1049) instance and each component `mongod` (page 1049) of the *sharded cluster*.

`connPoolStats.hosts.[host].available`

`available` (page 903) reports the total number of connections that the `mongos` (page 1061) or `mongod` (page 1049) could use to connect to this `mongod` (page 1049).

`connPoolStats.hosts.[host].created`

`created` (page 903) reports the number of connections that this `mongos` (page 1061) or `mongod` (page 1049) has ever created for this host.

`connPoolStats.replicaSets`

`replicaSets` (page 903) is a *document* that contains *replica set* information for the *sharded cluster*.

`connPoolStats.replicaSets.shard`

The `shard` (page 903) *document* reports on each *shard* within the *sharded cluster*

`connPoolStats.replicaSets.[shard].host`

The `host` (page 903) 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 865) values.

`connPoolStats.replicaSets.[shard].host[n].addr`

`addr` (page 903) reports the address for the host in the *sharded cluster* in the format of "`[hostname]:[port]`".

`connPoolStats.replicaSets.[shard].host[n].ok`

`ok` (page 903) reports `false` when:

- the `mongos` (page 1061) or `mongod` (page 1049) cannot connect to instance.
- the `mongos` (page 1061) or `mongod` (page 1049) received a connection exception or error.

This field is for internal use.

`connPoolStats.replicaSets.[shard].host[n].ismaster`

`ismaster` (page 903) reports `true` if this `host` (page 903) is the *primary* member of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].hidden`

`hidden` (page 903) reports `true` if this `host` (page 903) is a *hidden member* of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].secondary`

`secondary` (page 903) reports `true` if this `host` (page 903) is a *secondary* member of the *replica set*.

`connPoolStats.replicaSets.[shard].host[n].pingTimeMillis`

`pingTimeMillis` (page 903) reports the ping time in milliseconds from the `mongos` (page 1061) or `mongod` (page 1049) to this `host` (page 903).

`connPoolStats.replicaSets.[shard].host[n].tags`

New in version 2.2.

`tags` (page 903) reports the `tags` (page 475), if this member of the set has tags configured.

`connPoolStats.replicaSets.[shard].master`
`master` (page 904) reports the ordinal identifier of the host in the `host` (page 903) array that is the *primary* of the *replica set*.

`connPoolStats.replicaSets.[shard].nextSlave`
Deprecated since version 2.2.

`nextSlave` (page 904) reports the *secondary* member that the `mongos` (page 1061) will use to service the next request for this *replica set*.

`connPoolStats.createdByType`
`createdByType` (page 904) *document* reports the number of each type of connection that `mongos` (page 1061) or `mongod` (page 1049) has created in all connection pools.

`mongos` (page 1061) connect to `mongod` (page 1049) instances using one of three types of connections. The following sub-document reports the total number of connections by type.

`connPoolStats.createdByType.master`
`master` (page 904) reports the total number of connections to the *primary* member in each *cluster*.

`connPoolStats.createdByType.set`
`set` (page 904) reports the total number of connections to a *replica set* member.

`connPoolStats.createdByType.sync`
`sync` (page 904) reports the total number of *config database* connections.

`connPoolStats.totalAvailable`
`totalAvailable` (page 904) reports the running total of connections from the `mongos` (page 1061) or `mongod` (page 1049) to all `mongod` (page 1049) instances in the *sharded cluster* available for use.

`connPoolStats.totalCreated`
`totalCreated` (page 904) reports the total number of connections ever created from the `mongos` (page 1061) or `mongod` (page 1049) to all `mongod` (page 1049) instances in the *sharded cluster*.

`connPoolStats.numDBClientConnection`
`numDBClientConnection` (page 904) reports the total number of connections from the `mongos` (page 1061) or `mongod` (page 1049) to all of the `mongod` (page 1049) instances in the *sharded cluster*.

`connPoolStats.numAScopedConnection`
`numAScopedConnection` (page 904) reports the number of exception safe connections created from `mongos` (page 1061) or `mongod` (page 1049) to all `mongod` (page 1049) in the *sharded cluster*. The `mongos` (page 1061) or `mongod` (page 1049) releases these connections after receiving a socket exception from the `mongod` (page 1049).

dbStats

Definition

dbStats
The `dbStats` (page 904) command returns storage statistics for a given database. The command takes the following syntax:

```
{ dbStats: 1, scale: 1 }
```

The values of the options above do 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:

```
{ dbStats: 1, scale: 1024 }
```

Note: Because scaling rounds values to whole numbers, scaling may return unlikely or unexpected results.

The time required to run the command depends on the total size of the database. Because the command must touch all data files, the command may take several seconds to run.

In the `mongo` (page 1066) shell, the `db.stats()` (page 1013) function provides a wrapper around `dbStats` (page 904).

Output

`dbStats.`**`db`**
Contains the name of the database.

`dbStats.`**`collections`**
Contains a count of the number of collections in that database.

`dbStats.`**`objects`**
Contains a count of the number of objects (i.e. *documents*) in the database across all collections.

`dbStats.`**`avgObjSize`**
The average size of each document in bytes. This is the `dataSize` (page 905) divided by the number of documents.

`dbStats.`**`dataSize`**
The total size of the data held in this database including the *padding factor*. The `scale` argument affects this value. The `dataSize` (page 905) will not decrease when *documents* shrink, but will decrease when you remove documents.

`dbStats.`**`storageSize`**
The total amount of space allocated to collections in this database for *document* storage. The `scale` argument affects this value. The `storageSize` (page 905) does not decrease as you remove or shrink documents.

`dbStats.`**`numExtents`**
Contains a count of the number of extents in the database across all collections.

`dbStats.`**`indexes`**
Contains a count of the total number of indexes across all collections in the database.

`dbStats.`**`indexSize`**
The total size of all indexes created on this database. The `scale` arguments affects this value.

`dbStats.`**`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 905) only reflects the size of the data files for the database and not the namespace file.

The `scale` argument affects this value.

`dbStats.`**`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 1120) runtime option.

See also:

The `nssize` (page 1120) option, and *Maximum Namespace File Size* (page 1139)

`dbStats.`**`dataFileVersion`**
New in version 2.4.

Document that contains information about the on-disk format of the data files for the database.

dbStats.dataFileVersion.**major**
New in version 2.4.

The major version number for the on-disk format of the data files for the database.

dbStats.dataFileVersion.**minor**
New in version 2.4.

The minor version number for the on-disk format of the data files for the database.

cursorInfo

cursorInfo

The cursorInfo (page 906) command returns information about current cursor allotment and use. Use the following form:

```
{ cursorInfo: 1 }
```

The value (e.g. 1 above,) does not affect the output of the command.

cursorInfo (page 906) 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 (page 906) 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 (page 906) depends on the amount of data in the collection.

diagLogging

diagLogging

diagLogging (page 906) is an internal command.

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed.

getCmdLineOpts

getCmdLineOpts

The getCmdLineOpts (page 906) command returns a document containing command line options used to start the given mongod (page 1049):

```
{ 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 (page 1049). 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 (page 906):

```
{
        "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/

netstat

netstat

netstat (page 907) is an internal command that is only available on mongos (page 1061) instances.

ping

ping

The ping (page 907) 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.

profile

profile

Use the profile (page 907) 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. Your deployment should carefully consider the security implications of this. Consider the following prototype syntax:

```
{ profile: <level> }
```

The following profiling levels are available:

Level	Setting
-1	No change. Returns the current profile level.
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 (page 1049) writes the output of the database profiler to the system.profile collection.

mongod (page 1049) records queries that take longer than the slowms (page 1121) to the server log even when the database profiler is not active.

See also:

Additional documentation regarding database profiling *Database Profiling* (page 163).

See also:

"db.getProfilingStatus() (page 1006)" and "db.setProfilingLevel() (page 1012)" provide wrappers around this functionality in the mongo (page 1066) shell.

Note: The database cannot be locked with db.fsyncLock() (page 1004) while profiling is enabled. You must disable profiling before locking the database with db.fsyncLock() (page 1004). Disable profiling using db.setProfilingLevel() (page 1012) as follows in the mongo (page 1066) 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 held while enabling or disabling the profiler. This is typically a short operation.

validate

Definition

validate

The validate (page 908) command checks the structures within a namespace for correctness by scanning the collection's data and indexes. The command returns information regarding the on-disk representation of the collection.

The validate command can be slow, particularly on larger data sets.

The following example validates the contents of the collection named users.

```
{ validate: "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 (page 1066) 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.

Output

validate.**ns**
The full namespace name of the collection. Namespaces include the database name and the collection name in the form `database.collection`.

validate.**firstExtent**
The disk location of the first extent in the collection. The value of this field also includes the namespace.

validate.**lastExtent**
The disk location of the last extent in the collection. The value of this field also includes the namespace.

validate.**extentCount**
The number of extents in the collection.

validate.**extents**
`validate` (page 908) 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.

validate.extents.**loc**
The disk location for the beginning of this extent.

validate.extents.**xnext**
The disk location for the extent following this one. "null" if this is the end of the linked list of extents.

validate.extents.**xprev**
The disk location for the extent preceding this one. "null" if this is the head of the linked list of extents.

validate.extents.**nsdiag**
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

validate.extents.**size**
The number of bytes in this extent.

validate.extents.**firstRecord**
The disk location of the first record in this extent.

validate.extents.**lastRecord**
The disk location of the last record in this extent.

validate.**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 909) includes record *padding*.

validate.**nrecords**
The number of *documents* in the collection.

validate.**lastExtentSize**
The size of the last new extent created in this collection. This value determines the size of the *next* extent created.

validate.**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.

validate.**firstExtentDetails**
The size of the first extent created in this collection. This data is similar to the data provided by the `extents` (page 909) sub-document; however, the data reflects only the first extent in the collection and is always returned.

validate.firstExtentDetails.**loc**
The disk location for the beginning of this extent.

validate.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.

validate.firstExtentDetails.**xprev**
The disk location for the extent preceding this one. This should always be "null."

validate.firstExtentDetails.**nsdiag**
The namespace this extent belongs to (should be the same as the namespace shown at the beginning of the validate listing).

validate.firstExtentDetails.**size**
The number of bytes in this extent.

validate.firstExtentDetails.**firstRecord**
The disk location of the first record in this extent.

validate.firstExtentDetails.**lastRecord**
The disk location of the last record in this extent.

validate.**objectsFound**
The number of records actually encountered in a scan of the collection. This field should have the same value as the `nrecords` (page 909) field.

validate.**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.

validate.**bytesWithHeaders**
This is similar to datasize, except that `bytesWithHeaders` (page 910) 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.

validate.**bytesWithoutHeaders**
`bytesWithoutHeaders` (page 910) returns data collected from a scan of all records. The value should be the same as `datasize` (page 909).

Note: This field is only included in the validation output when you specify the `full` option.

validate.**deletedCount**
The number of deleted or "free" records in the collection.

validate.**deletedSize**
The size of all deleted or "free" records in the collection.

validate.**nIndexes**
The number of indexes on the data in the collection.

validate.**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.

validate.**valid**
Boolean. `true`, unless `validate` (page 908) determines that an aspect of the collection is not valid. When

`false`, see the `errors` (page 911) field for more information.

validate.errors
Typically empty; however, if the collection is not valid (i.e `valid` (page 910) is false,) this field will contain a message describing the validation error.

validate.ok
Set to `1` when the command succeeds. If the command fails the `ok` (page 911) field has a value of `0`.

top

top
The `top` (page 911) 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` (page 911) command against the *admin database* in the form:

```
{ top: 1 }
```

indexStats

Definition

indexStats
The `indexStats` (page 911) command aggregates statistics for the B-tree data structure that stores data for a MongoDB index.

> **Warning:** This command is not intended for production deployments.

The command can be run *only* on a `mongod` (page 1049) instance that uses the `--enableExperimentalIndexStatsCmd` option.

To aggregate statistics, issue the command like so:

```
db.runCommand( { indexStats: "<collection>", index: "<index name>" } )
```

Output The `db.collection.indexStats()` method and equivalent `indexStats` (page 911) command aggregate statistics for the B-tree data structure that stores data for a MongoDB index. The commands aggregate statistics firstly for the entire B-tree and secondly for each individual level of the B-tree. The output displays the following values.

`indexStats.`**`index`**
The *index name* (page 315).

`indexStats.`**`version`**
The index version. For more information on index version numbers, see the `v` option in `db.collection.ensureIndex()` (page 949).

`indexStats.`**`isIdIndex`**
If `true`, the index is the default `_id` index for the collection.

`indexStats.`**`keyPattern`**
The indexed keys.

`indexStats.`**`storageNs`**
The namespace of the index's underlying storage.

`indexStats.`**`bucketBodyBytes`**
The fixed size, in bytes, of a B-tree bucket in the index, not including the record header. All indexes for a given version have the same value for this field. MongoDB allocates fixed size buckets on disk.

`indexStats.`**`depth`**
The number of levels in the B-tree, not including the root level.

`indexStats.`**`overall`**
This section of the output displays statistics for the entire B-tree.

`indexStats.overall.`**`numBuckets`**
The number of buckets in the entire B-tree, including all levels.

`indexStats.overall.`**`keyCount`**
Statistics about the number of keys in a bucket, evaluated on a per-bucket level.

`indexStats.overall.`**`usedKeyCount`**
Statistics about the number of used keys in a bucket, evaluated on a per-bucket level. Used keys are keys not marked as deleted.

`indexStats.overall.`**`bsonRatio`**
Statistics about the percentage of the bucket body that is occupied by the key objects themselves, excluding associated metadata.

For example, if you have the document `{ name: "Bob Smith" }` and an index on `{ name: 1 }`, the key object is the string `Bob Smith`.

`indexStats.overall.`**`keyNodeRatio`**
Statistics about the percentage of the bucket body that is occupied by the key node objects (the metadata and links pertaining to the keys). This does not include the key itself. In the current implementation, a key node's objects consist of: the pointer to the key data (in the same bucket), the pointer to the record the key is for, and the pointer to a child bucket.

`indexStats.overall.`**`fillRatio`**
The sum of the `bsonRatio` (page 912) and the `keyNodeRatio` (page 912). This shows how full the buckets are. This will be much higher for indexes with sequential inserts.

`indexStats.`**`perLevel`**
This section of the output displays statistics for each level of the B-tree separately, starting with the root level. This section displays a different document for each B-tree level.

indexStats.perLevel.**numBuckets**
The number of buckets at this level of the B-tree.

indexStats.perLevel.**keyCount**
Statistics about the number of keys in a bucket, evaluated on a per-bucket level.

indexStats.perLevel.**usedKeyCount**
Statistics about the number of used keys in a bucket, evaluated on a per-bucket level. Used keys are keys not marked as deleted.

indexStats.perLevel.**bsonRatio**
Statistics about the percentage of the bucket body that is occupied by the key objects themselves, excluding associated metadata.

indexStats.perLevel.**keyNodeRatio**
Statistics about the percentage of the bucket body that is occupied by the key node objects (the metadata and links pertaining to the keys).

indexStats.perLevel.**fillRatio**
The sum of the bsonRatio (page 913) and the keyNodeRatio (page 913). This shows how full the buckets are. This will be much higher in the following cases:

- For indexes with sequential inserts, such as the _id index when using ObjectId keys.
- For indexes that were recently built in the foreground with existing data.
- If you recently ran compact (page 890) or *--repair*.

Example The following is an example of db.collection.indexStats() and indexStats (page 911) output.

```
{
    "index" : "type_1_traits_1",
    "version" : 1,
    "isIdIndex" : false,
    "keyPattern" : {
        "type" : 1,
        "traits" : 1
    },
    "storageNs" : "test.animals.$type_1_traits_1",
    "bucketBodyBytes" : 8154,
    "depth" : 2,
    "overall" : {
        "numBuckets" : 45513,
        "keyCount" : {
            "count" : NumberLong(45513),
            "mean" : 253.89602970579836,
            "stddev" : 21.784799875240708,
            "min" : 52,
            "max" : 290,
            "quantiles" : {
                "0.01" : 201.99785091648775,
                // ...
                "0.99" : 289.9999655156967
            }
        },
        "usedKeyCount" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
```

```
                "0.01" : 201.99785091648775,
                // ...
                "0.99" : 289.9999655156967
            }
        },
        "bsonRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.4267797891997124,
                // ...
                "0.99" : 0.5945548174629648
            }
        },
        "keyNodeRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.3963656628236211,
                // ...
                "0.99" : 0.5690457993930765
            }
        },
        "fillRatio" : {
            "count" : NumberLong(45513),
            // ...
            "quantiles" : {
                "0.01" : 0.9909134214926929,
                // ...
                "0.99" : 0.9960755457453732
            }
        }
    },
    "perLevel" : [
        {
            "numBuckets" : 1,
            "keyCount" : {
                "count" : NumberLong(1),
                "mean" : 180,
                "stddev" : 0,
                "min" : 180,
                "max" : 180
            },
            "usedKeyCount" : {
                "count" : NumberLong(1),
                // ...
                "max" : 180
            },
            "bsonRatio" : {
                "count" : NumberLong(1),
                // ...
                "max" : 0.3619082658817758
            },
            "keyNodeRatio" : {
                "count" : NumberLong(1),
                // ...
                "max" : 0.35320088300220753
            },
```

```
        "fillRatio" : {
            "count" : NumberLong(1),
            // ...
            "max" : 0.7151091488839834
        }
    },
    {
        "numBuckets" : 180,
        "keyCount" : {
            "count" : NumberLong(180),
            "mean" : 250.84444444444443,
            "stddev" : 26.30057503009355,
            "min" : 52,
            "max" : 290
        },
        "usedKeyCount" : {
            "count" : NumberLong(180),
            // ...
            "max" : 290
        },
        "bsonRatio" : {
            "count" : NumberLong(180),
            // ...
            "max" : 0.5945548197203826
        },
        "keyNodeRatio" : {
            "count" : NumberLong(180),
            // ...
            "max" : 0.5690458670591121
        },
        "fillRatio" : {
            "count" : NumberLong(180),
            // ...
            "max" : 0.9963208241353937
        }
    },
    {
        "numBuckets" : 45332,
        "keyCount" : {
            "count" : NumberLong(45332),
            "mean" : 253.90977675813994,
            "stddev" : 21.761620836279018,
            "min" : 167,
            "max" : 290,
            "quantiles" : {
                "0.01" : 202.0000012563603,
                // ...
                "0.99" : 289.99996486571894
            }
        },
        "usedKeyCount" : {
            "count" : NumberLong(45332),
            // ...
            "quantiles" : {
                "0.01" : 202.0000012563603,
                // ...
                "0.99" : 289.99996486571894
            }
```

```
            },
            "bsonRatio" : {
                "count" : NumberLong(45332),
                // ...
                "quantiles" : {
                    "0.01" : 0.42678446958950583,
                    // ...
                    "0.99" : 0.5945548175411283
                }
            },
            "keyNodeRatio" : {
                "count" : NumberLong(45332),
                // ...
                "quantiles" : {
                    "0.01" : 0.39636988227885306,
                    // ...
                    "0.99" : 0.5690457981176729
                }
            },
            "fillRatio" : {
                "count" : NumberLong(45332),
                // ...
                "quantiles" : {
                    "0.01" : 0.9909246995605362,
                    // ...
                    "0.99" : 0.996075546919481
                }
            }
        }
    ],
    "ok" : 1
}
```

Additional Resources For more information on the command's limits and output, see the following:

- The equivalent `db.collection.indexStats()` method,
- *indexStats* (page 911), and
- https://github.com/10gen-labs/storage-viz#readme.

whatsmyuri

whatsmyuri

`whatsmyuri` (page 916) is an internal command.

getLog

getLog

The `getLog` (page 916) command returns a document with a `log` array that contains recent messages from the `mongod` (page 1049) process log. The `getLog` (page 916) 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` (page 1049) is part of a *replica set*, `getLog` (page 916) 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` (page 1049) 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` (page 1066) shell connected to a replica set:

```
db.adminCommand({getLog: "*" })
{ "names" : [ "global", "rs", "startupWarnings" ], "ok" : 1 }
```

`getLog` (page 916) returns events from a RAM cache of the `mongod` (page 1049) events and *does not* read log data from the log file.

hostInfo

`hostInfo`

New in version 2.2.

Returns A document with information about the underlying system that the `mongod` (page 1049) or `mongos` (page 1061) runs on. Some of the returned fields are only included on some platforms.

You must run the `hostInfo` (page 917) command, which takes no arguments, against the `admin` database. Consider the following invocations of `hostInfo` (page 917):

```
db.hostInfo()
db.adminCommand( { "hostInfo" : 1 } )
```

In the `mongo` (page 1066) shell you can use `db.hostInfo()` (page 1008) as a helper to access `hostInfo` (page 917). The output of `hostInfo` (page 917) on a Linux system will resemble the following:

```
{
   "system" : {
          "currentTime" : ISODate("<timestamp>"),
          "hostname" : "<hostname>",
          "cpuAddrSize" : <number>,
          "memSizeMB" : <number>,
          "numCores" : <number>,
          "cpuArch" : "<identifier>",
          "numaEnabled" : <boolean>
   },
   "os" : {
          "type" : "<string>",
          "name" : "<string>",
          "version" : "<string>"
   },
   "extra" : {
          "versionString" : "<string>",
          "libcVersion" : "<string>",
          "kernelVersion" : "<string>",
          "cpuFrequencyMHz" : "<string>",
          "cpuFeatures" : "<string>",
          "pageSize" : <number>,
          "numPages" : <number>,
          "maxOpenFiles" : <number>
   },
   "ok" : <return>
}
```

Consider the following documentation of these fields:

hostInfo
The document returned by the `hostInfo` (page 917).

hostInfo.**system**
A sub-document about the underlying environment of the system running the `mongod` (page 1049) or `mongos` (page 1061)

hostInfo.system.**currentTime**
A time stamp of the current system time.

hostInfo.system.**hostname**
The system name, which should correspond to the output of `hostname -f` on Linux systems.

hostInfo.system.**cpuAddrSize**
A number reflecting the architecture of the system. Either `32` or `64`.

hostInfo.system.**memSizeMB**
The total amount of system memory (RAM) in megabytes.

hostInfo.system.**numCores**
The total number of available logical processor cores.

hostInfo.system.**cpuArch**
A string that represents the system architecture. Either `x86` or `x86_64`.

hostInfo.system.**numaEnabled**
A boolean value. `false` if NUMA is interleaved (i.e. disabled,) otherwise `true`.

hostInfo.**os**
A sub-document that contains information about the operating system running the `mongod` (page 1049) and `mongos` (page 1061).

hostInfo.os.**type**
A string representing the type of operating system, such as `Linux` or `Windows`.

hostInfo.os.**name**
If available, returns a display name for the operating system.

hostInfo.os.**version**
If available, returns the name of the distribution or operating system.

hostInfo.**extra**
A sub-document with extra information about the operating system and the underlying hardware. The content of the `extra` (page 918) sub-document depends on the operating system.

hostInfo.extra.**versionString**
A complete string of the operating system version and identification. On Linux and OS X systems, this contains output similar to `uname -a`.

hostInfo.extra.**libcVersion**
The release of the system `libc`.

`libcVersion` (page 918) only appears on Linux systems.

hostInfo.extra.**kernelVersion**
The release of the Linux kernel in current use.

`kernelVersion` (page 918) only appears on Linux systems.

hostInfo.extra.**alwaysFullSync**
`alwaysFullSync` (page 918) only appears on OS X systems.

hostInfo.extra.**nfsAsync**
nfsAsync (page 918) only appears on OS X systems.

hostInfo.extra.**cpuFrequencyMHz**
Reports the clock speed of the system's processor in megahertz.

hostInfo.extra.**cpuFeatures**
Reports the processor feature flags. On Linux systems this the same information that `http://docs.mongodb.org/manualproc/cpuinfo` includes in the `flags` fields.

hostInfo.extra.**pageSize**
Reports the default system page size in bytes.

hostInfo.extra.**numPages**
numPages (page 919) only appears on Linux systems.

hostInfo.extra.**maxOpenFiles**
Reports the current system limits on open file handles. See *UNIX ulimit Settings* (page 193) for more information.

maxOpenFiles (page 919) only appears on Linux systems.

hostInfo.extra.**scheduler**
Reports the active I/O scheduler. scheduler (page 919) only appears on OS X systems.

serverStatus

Definition

serverStatus
The serverStatus (page 919) 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.

Changed in version 2.4: In 2.4 you can dynamically suppress portions of the serverStatus (page 919) output, or include suppressed sections by adding fields to the command document as in the following examples:

```
db.runCommand( { serverStatus: 1, repl: 0, indexCounters: 0 } )
db.runCommand( { serverStatus: 1, workingSet: 1, metrics: 0, locks: 0 } )
```

serverStatus (page 919) includes all fields by default, except workingSet (page 932), by default.

Note: You may only dynamically include top-level fields from the *serverStatus* (page 919) document that are not included by default. You can exclude any field that serverStatus (page 919) includes by default.

See also:

db.serverStatus() (page 1012) and "`http://docs.mongodb.org/manualreference/server-status`"

Output The serverStatus (page 919) 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` (page 919) is also displayed dynamically by `mongostat` (page 1098). See the *mongostat* (page 1097) command for more information.

For examples of the `serverStatus` (page 919) output, see `http://docs.mongodb.org/manualreference/server-sta`

Instance Information For an example of the instance information, see the *Instance Information section* of the `http://docs.mongodb.org/manualreference/server-status` page.

serverStatus.**host**

The `host` (page 920) field contains the system's hostname. In Unix/Linux systems, this should be the same as the output of the `hostname` command.

serverStatus.**version**

The `version` (page 920) field contains the version of MongoDB running on the current `mongod` (page 1049) or `mongos` (page 1061) instance.

serverStatus.**process**

The `process` (page 920) field identifies which kind of MongoDB instance is running. Possible values are:

- `mongos` (page 1061)
- `mongod` (page 1049)

serverStatus.**uptime**

The value of the `uptime` (page 920) field corresponds to the number of seconds that the `mongos` (page 1061) or `mongod` (page 1049) process has been active.

serverStatus.**uptimeEstimate**

`uptimeEstimate` (page 920) provides the uptime as calculated from MongoDB's internal course-grained time keeping system.

serverStatus.**localTime**

The `localTime` (page 920) value is the current time, according to the server, in UTC specified in an ISODate format.

locks New in version 2.1.2: All `locks` (page 920) statuses first appeared in the 2.1.2 development release for the 2.2 series.

For an example of the `locks` output, see the *locks section* of the `http://docs.mongodb.org/manualreference/server-status` page.

serverStatus.**locks**

The `locks` (page 920) 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` (page 1049) started.

serverStatus.locks..

A field named `.` holds the first document in `locks` (page 920) that contains information about the global lock.

serverStatus.locks...**timeLockedMicros**
The timeLockedMicros (page 920) document reports the amount of time in microseconds that a lock has existed in all databases in this mongod (page 1049) instance.

serverStatus.locks...timeLockedMicros.**R**
The R field reports the amount of time in microseconds that any database has held the global read lock.

serverStatus.locks...timeLockedMicros.**W**
The W field reports the amount of time in microseconds that any database has held the global write lock.

serverStatus.locks...timeLockedMicros.**r**
The r field reports the amount of time in microseconds that any database has held the local read lock.

serverStatus.locks...timeLockedMicros.**w**
The w field reports the amount of time in microseconds that any database has held the local write lock.

serverStatus.locks...**timeAcquiringMicros**
The timeAcquiringMicros (page 921) document reports the amount of time in microseconds that operations have spent waiting to acquire a lock in all databases in this mongod (page 1049) instance.

serverStatus.locks...timeAcquiringMicros.**R**
The R field reports the amount of time in microseconds that any database has spent waiting for the global read lock.

serverStatus.locks...timeAcquiringMicros.**W**
The W field reports the amount of time in microseconds that any database has spent waiting for the global write lock.

serverStatus.locks.**admin**
The admin (page 921) document contains two sub-documents that report data regarding lock use in the *admin database*.

serverStatus.locks.admin.**timeLockedMicros**
The timeLockedMicros (page 921) document reports the amount of time in microseconds that locks have existed in the context of the *admin database*.

serverStatus.locks.admin.timeLockedMicros.**r**
The r field reports the amount of time in microseconds that the *admin database* has held the read lock.

serverStatus.locks.admin.timeLockedMicros.**w**
The w field reports the amount of time in microseconds that the *admin database* has held the write lock.

serverStatus.locks.admin.**timeAcquiringMicros**
The timeAcquiringMicros (page 921) document reports on the amount of field time in microseconds that operations have spent waiting to acquire a lock for the *admin database*.

serverStatus.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*.

serverStatus.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*.

serverStatus.locks.**local**
The local (page 921) 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.

serverStatus.locks.local.**timeLockedMicros**
The timeLockedMicros (page 921) document reports on the amount of time in microseconds that locks have existed in the context of the local database.

serverStatus.locks.local.timeLockedMicros.**r**
The r field reports the amount of time in microseconds that the local database has held the read lock.

serverStatus.locks.local.timeLockedMicros.**w**
The w field reports the amount of time in microseconds that the local database has held the write lock.

serverStatus.locks.local.**timeAcquiringMicros**
The timeAcquiringMicros (page 922) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the local database.

serverStatus.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.

serverStatus.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.

serverStatus.locks.**<database>**
For each additional database locks (page 920) includes a document that reports on the lock use for this database. The names of these documents reflect the database name itself.

serverStatus.locks.<database>.**timeLockedMicros**
The timeLockedMicros (page 922) document reports on the amount of time in microseconds that locks have existed in the context of the <database> database.

serverStatus.locks.<database>.timeLockedMicros.**r**
The r field reports the amount of time in microseconds that the <database> database has held the read lock.

serverStatus.locks.<database>.timeLockedMicros.**w**
The w field reports the amount of time in microseconds that the <database> database has held the write lock.

serverStatus.locks.<database>.**timeAcquiringMicros**
The timeAcquiringMicros (page 922) document reports on the amount of time in microseconds that operations have spent waiting to acquire a lock for the <database> database.

serverStatus.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.

serverStatus.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.

globalLock For an example of the globalLock output, see the *globalLock section* of the http://docs.mongodb.org/manualreference/server-status page.

serverStatus.**globalLock**
The globalLock (page 922) data structure contains information regarding the database's current lock state, historical lock status, current operation queue, and the number of active clients.

serverStatus.globalLock.**totalTime**
The value of totalTime (page 922) represents the time, in microseconds, since the database last started and creation of the globalLock (page 922). This is roughly equivalent to total server uptime.

serverStatus.globalLock.**lockTime**
The value of lockTime (page 922) represents the time, in microseconds, since the database last started, that the globalLock (page 922) has been *held*.

Consider this value in combination with the value of totalTime (page 922). MongoDB aggregates these values in the ratio (page 923) value. If the ratio (page 923) value is small but totalTime (page 922) is high the `globalLock` (page 922) has typically been held frequently for shorter periods of time, which may be indicative of a more normal use pattern. If the `lockTime` (page 922) is higher and the `totalTime` (page 922) is smaller (relatively,) then fewer operations are responsible for a greater portion of server's use (relatively.)

`serverStatus.globalLock.`**`ratio`**

Changed in version 2.2: `ratio` (page 923) was removed. See `locks` (page 920).

The value of `ratio` (page 923) displays the relationship between `lockTime` (page 922) and `totalTime` (page 922).

Low values indicate that operations have held the `globalLock` (page 922) frequently for shorter periods of time. High values indicate that operations have held `globalLock` (page 922) infrequently for longer periods of time.

`serverStatus.globalLock.`**`currentQueue`**

The `currentQueue` (page 923) data structure value provides more granular information concerning the number of operations queued because of a lock.

`serverStatus.globalLock.currentQueue.`**`total`**

The value of `total` (page 923) 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. `readers` (page 923)) and write lock (e.g. `writers` (page 923)) individually.

`serverStatus.globalLock.currentQueue.`**`readers`**

The value of `readers` (page 923) 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.

`serverStatus.globalLock.currentQueue.`**`writers`**

The value of `writers` (page 923) 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

`serverStatus.globalLock.`**`activeClients`**

The `activeClients` (page 923) 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 923) data.

`serverStatus.globalLock.activeClients.`**`total`**

The value of `total` (page 923) is the total number of active client connections to the database. This combines clients that are performing read operations (e.g. `readers` (page 923)) and clients that are performing write operations (e.g. `writers` (page 923)).

`serverStatus.globalLock.activeClients.`**`readers`**

The value of `readers` (page 923) contains a count of the active client connections performing read operations.

`serverStatus.globalLock.activeClients.`**`writers`**

The value of `writers` (page 923) contains a count of active client connections performing write operations.

mem For an example of the `mem` output, see the *mem section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.`**`mem`**

The `mem` (page 923) data structure holds information regarding the target system architecture of `mongod` (page 1049) and current memory use.

`serverStatus.mem.`**`bits`**

The value of `bits` (page 923) is either `64` or `32`, depending on which target architecture specified during the mongod (page 1049) compilation process. In most instances this is 64, and this value does not change over time.

serverStatus.mem.**resident**
The value of resident (page 924) is roughly equivalent to the amount of RAM, in megabytes (MB), 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.

serverStatus.mem.**virtual**
virtual (page 924) displays the quantity, in megabytes (MB), of virtual memory used by the mongod (page 1049) process. With *journaling* enabled, the value of virtual (page 924) is at least twice the value of mapped (page 924).

If virtual (page 924) value is significantly larger than mapped (page 924) (e.g. 3 or more times), this may indicate a memory leak.

serverStatus.mem.**supported**
supported (page 924) 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 (page 923) values may not be accessible to the database server.

serverStatus.mem.**mapped**
The value of mapped (page 924) provides the amount of mapped memory, in megabytes (MB), 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.

serverStatus.mem.**mappedWithJournal**
mappedWithJournal (page 924) provides the amount of mapped memory, in megabytes (MB), including the memory used for journaling. This value will always be twice the value of mapped (page 924). This field is only included if journaling is enabled.

connections For an example of the connections output, see the *connections section* of the http://docs.mongodb.org/manualreference/server-status page.

serverStatus.**connections**
The connections (page 924) 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.

serverStatus.connections.**current**
The value of current (page 924) corresponds to the number of connections to the database server from clients. This number includes the current shell session. Consider the value of available (page 924) 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*.

serverStatus.connections.**available**
available (page 924) provides a count of the number of unused available connections that the database can provide. Consider this value in combination with the value of current (page 924) to understand the connection load on the database, and the *UNIX ulimit Settings* (page 193) document for more information about system thresholds on available connections.

serverStatus.connections.**totalCreated**
totalCreated (page 924) provides a count of **all** connections created to the server. This number includes connections that have since closed.

extra_info For an example of the extra_info output, see the *extra_info section* of the http://docs.mongodb.org/manualreference/server-status page.

`serverStatus.`**`extra_info`**
The `extra_info` (page 924) data structure holds data collected by the `mongod` (page 1049) instance about the underlying system. Your system may only report a subset of these fields.

`serverStatus.extra_info.`**`note`**
The field `note` (page 925) reports that the data in this structure depend on the underlying platform, and has the text: "fields vary by platform."

`serverStatus.extra_info.`**`heap_usage_bytes`**
The `heap_usage_bytes` (page 925) field is only available on Unix/Linux systems, and reports the total size in bytes of heap space used by the database process.

`serverStatus.extra_info.`**`page_faults`**
The `page_faults` (page 925) 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_faults` (page 925) 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.

indexCounters For an example of the `indexCounters` output, see the *indexCounters section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.`**`indexCounters`**
Changed in version 2.2: Previously, data in the `indexCounters` (page 925) 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.

Changed in version 2.4: Fields previously in the `btree` sub-document of `indexCounters` (page 925) are now fields in the `indexCounters` (page 925) document.

The `indexCounters` (page 925) data structure reports information regarding the state and use of indexes in MongoDB.

`serverStatus.indexCounters.`**`accesses`**
`accesses` (page 925) reports the number of times that operations have accessed indexes. This value is the combination of the `hits` (page 925) and `misses` (page 925). 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.

`serverStatus.indexCounters.`**`hits`**
The `hits` (page 925) value reflects the number of times that an index has been accessed and `mongod` (page 1049) is able to return the index from memory.

A higher value indicates effective index use. `hits` (page 925) values that represent a greater proportion of the `accesses` (page 925) value, tend to indicate more effective index configuration.

`serverStatus.indexCounters.`**`misses`**
The `misses` (page 925) 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.

`serverStatus.indexCounters.`**`resets`**
The `resets` (page 925) 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 925) values.

`serverStatus.indexCounters.`**`missRatio`**
The `missRatio` (page 925) value is the ratio of `hits` (page 925) to `misses` (page 925) misses. This value is typically `0` or approaching `0`.

backgroundFlushing For an example of the `backgroundFlushing` output, see the *backgroundFlushing section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.`**`backgroundFlushing`**

`mongod` (page 1049) periodically flushes writes to disk. In the default configuration, this happens every 60 seconds. The `backgroundFlushing` (page 926) data structure contains data regarding these operations. Consider these values if you have concerns about write performance and *journaling* (page 930).

`serverStatus.backgroundFlushing.`**`flushes`**

`flushes` (page 926) 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.

`serverStatus.backgroundFlushing.`**`total_ms`**

The `total_ms` (page 926) value provides the total number of milliseconds (ms) that the `mongod` (page 1049) processes have spent writing (i.e. flushing) data to disk. Because this is an absolute value, consider the value of `flushes` (page 926) and `average_ms` (page 926) to provide better context for this datum.

`serverStatus.backgroundFlushing.`**`average_ms`**

The `average_ms` (page 926) 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 `flushes` (page 926) is, the more likely this value is likely to represent a "normal," time; however, abnormal data can skew this value.

Use the `last_ms` (page 926) to ensure that a high average is not skewed by transient historical issue or a random write distribution.

`serverStatus.backgroundFlushing.`**`last_ms`**

The value of the `last_ms` (page 926) 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 `average_ms` (page 926) and `total_ms` (page 926).

`serverStatus.backgroundFlushing.`**`last_finished`**

The `last_finished` (page 926) 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.

cursors For an example of the `cursors` output, see the *cursors section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.`**`cursors`**

The `cursors` (page 926) data structure contains data regarding cursor state and use.

`serverStatus.cursors.`**`totalOpen`**

`totalOpen` (page 926) 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.

`serverStatus.cursors.`**`clientCursors_size`**

Deprecated since version 1.x: See `totalOpen` (page 926) for this datum.

`serverStatus.cursors.`**`timedOut`**

`timedOut` (page 926) 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.

network For an example of the `network` output, see the *network section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.network`
The `network` (page 926) data structure contains data regarding MongoDB's network use.

`serverStatus.network.bytesIn`
The value of the `bytesIn` (page 927) 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` (page 1049) process is consistent with expectations and overall inter-application traffic.

`serverStatus.network.bytesOut`
The value of the `bytesOut` (page 927) 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` (page 1049) process is consistent with expectations and overall inter-application traffic.

`serverStatus.network.numRequests`
The `numRequests` (page 927) field is a counter of the total number of distinct requests that the server has received. Use this value to provide context for the `bytesIn` (page 927) and `bytesOut` (page 927) values to ensure that MongoDB's network utilization is consistent with expectations and application use.

repl For an example of the `repl` output, see the *repl section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.repl`
The `repl` (page 927) 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* (page 367) for more information on replication.

`serverStatus.repl.setName`
The `setName` (page 927) field contains a string with the name of the current replica set. This value reflects the `--replSet` command line argument, or `replSet` (page 1124) value in the configuration file.

See *Replication* (page 367) for more information on replication.

`serverStatus.repl.ismaster`
The value of the `ismaster` (page 927) field is either `true` or `false` and reflects whether the current node is the master or primary node in the replica set.

See *Replication* (page 367) for more information on replication.

`serverStatus.repl.secondary`
The value of the `secondary` (page 927) field is either `true` or `false` and reflects whether the current node is a secondary node in the replica set.

See *Replication* (page 367) for more information on replication.

`serverStatus.repl.hosts`
`hosts` (page 927) 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* (page 367) for more information on replication.

opcountersRepl For an example of the `opcountersRepl` output, see the *opcountersRepl section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.opcountersRepl`
The `opcountersRepl` (page 927) data structure, similar to the `opcounters` (page 928) 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` (page 928) values because of how MongoDB serializes operations during replication. See *Replication* (page 367) 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.

`serverStatus.opcountersRepl.`**`insert`**
`insert` (page 928) provides a counter of the total number of replicated insert operations since the `mongod` (page 1049) instance last started.

`serverStatus.opcountersRepl.`**`query`**
`query` (page 928) provides a counter of the total number of replicated queries since the `mongod` (page 1049) instance last started.

`serverStatus.opcountersRepl.`**`update`**
`update` (page 928) provides a counter of the total number of replicated update operations since the `mongod` (page 1049) instance last started.

`serverStatus.opcountersRepl.`**`delete`**
`delete` (page 928) provides a counter of the total number of replicated delete operations since the `mongod` (page 1049) instance last started.

`serverStatus.opcountersRepl.`**`getmore`**
`getmore` (page 928) provides a counter of the total number of "getmore" operations since the `mongod` (page 1049) 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.

`serverStatus.opcountersRepl.`**`command`**
`command` (page 928) provides a counter of the total number of replicated commands issued to the database since the `mongod` (page 1049) instance last started.

opcounters For an example of the `opcounters` output, see the *opcounters section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.`**`opcounters`**
The `opcounters` (page 928) 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.

Note: The data in `opcounters` (page 928) treats operations that affect multiple documents, such as bulk insert or multi-update operations, as a single operation. See `document` (page 932) for more granular document-level operation tracking.

`serverStatus.opcounters.`**`insert`**
`insert` (page 928) provides a counter of the total number of insert operations since the `mongod` (page 1049) instance last started.

`serverStatus.opcounters.`**`query`**
`query` (page 928) provides a counter of the total number of queries since the `mongod` (page 1049) instance last started.

`serverStatus.opcounters.`**`update`**
`update` (page 928) provides a counter of the total number of update operations since the `mongod` (page 1049) instance last started.

serverStatus.opcounters.**delete**

delete (page 928) provides a counter of the total number of delete operations since the mongod (page 1049) instance last started.

serverStatus.opcounters.**getmore**

getmore (page 929) provides a counter of the total number of "getmore" operations since the mongod (page 1049) 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.

serverStatus.opcounters.**command**

command (page 929) provides a counter of the total number of commands issued to the database since the mongod (page 1049) instance last started.

asserts For an example of the asserts output, see the *asserts section* of the http://docs.mongodb.org/manualreference/server-status page.

serverStatus.**asserts**

The asserts (page 929) document reports the number of asserts on the database. While assert errors are typically uncommon, if there are non-zero values for the asserts (page 929), you should check the log file for the mongod (page 1049) process for more information. In many cases these errors are trivial, but are worth investigating.

serverStatus.asserts.**regular**

The regular (page 929) counter tracks the number of regular assertions raised since the server process started. Check the log file for more information about these messages.

serverStatus.asserts.**warning**

The warning (page 929) counter tracks the number of warnings raised since the server process started. Check the log file for more information about these warnings.

serverStatus.asserts.**msg**

The msg (page 929) counter tracks the number of message assertions raised since the server process started. Check the log file for more information about these messages.

serverStatus.asserts.**user**

The user (page 929) 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.

serverStatus.asserts.**rollovers**

The rollovers (page 929) 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 (page 929) data structure.

writeBacksQueued For an example of the writeBacksQueued output, see the *writeBacksQueued section* of the http://docs.mongodb.org/manualreference/server-status page.

serverStatus.**writeBacksQueued**

The value of writeBacksQueued (page 929) is true when there are operations from a mongos (page 1061) instance queued for retrying. Typically this option is false.

See also:

writeBacks

Journaling (dur) New in version 1.8.

For an example of the `Journaling (dur)` output, see the *journaling section* of the `http://docs.mongodb.org/manualreference/server-status` page.

`serverStatus.dur`
The `dur` (page 930) (for "durability") document contains data regarding the `mongod` (page 1049)'s journaling-related operations and performance. `mongod` (page 1049) must be running with journaling for these data to appear in the output of "`serverStatus` (page 919)".

Note: The data values are **not** cumulative but are reset on a regular basis as determined by the *journal group commit interval* (page 156). 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 2/3 when there is a `getLastError` (page 861) command pending. The interval is configurable using the `--journalCommitInterval` option.

See also:

"*Journaling* (page 155)" for more information about journaling operations.

`serverStatus.dur.commits`
The `commits` (page 930) provides the number of transactions written to the *journal* during the last *journal group commit interval* (page 156).

`serverStatus.dur.journaledMB`
The `journaledMB` (page 930) provides the amount of data in megabytes (MB) written to *journal* during the last *journal group commit interval* (page 158).

`serverStatus.dur.writeToDataFilesMB`
The `writeToDataFilesMB` (page 930) provides the amount of data in megabytes (MB) written from *journal* to the data files during the last *journal group commit interval* (page 158).

`serverStatus.dur.compression`
New in version 2.0.

The `compression` (page 930) represents the compression ratio of the data written to the *journal*:

```
( journaled_size_of_data / uncompressed_size_of_data )
```

`serverStatus.dur.commitsInWriteLock`
The `commitsInWriteLock` (page 930) 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.

`serverStatus.dur.earlyCommits`
The `earlyCommits` (page 930) value reflects the number of times MongoDB requested a commit before the scheduled *journal group commit interval* (page 158). Use this value to ensure that your *journal group commit interval* (page 156) is not too long for your deployment.

`serverStatus.dur.timeMS`
The `timeMS` (page 930) document provides information about the performance of the `mongod` (page 1049) instance during the various phases of journaling in the last *journal group commit interval* (page 156).

`serverStatus.dur.timeMS.dt`
The `dt` (page 930) value provides, in milliseconds, the amount of time over which MongoDB collected the `timeMS` (page 930) data. Use this field to provide context to the other `timeMS` (page 930) field values.

`serverStatus.dur.timeMS.prepLogBuffer`
The `prepLogBuffer` (page 930) value provides, in milliseconds, the amount of time spent preparing to write to the journal. Smaller values indicate better journal performance.

serverStatus.dur.timeMS.**writeToJournal**

The `writeToJournal` (page 930) value provides, in milliseconds, the amount of time spent actually writing to the journal. File system speeds and device interfaces can affect performance.

serverStatus.dur.timeMS.**writeToDataFiles**

The `writeToDataFiles` (page 931) value provides, in milliseconds, the amount of time spent writing to data files after journaling. File system speeds and device interfaces can affect performance.

serverStatus.dur.timeMS.**remapPrivateView**

The `remapPrivateView` (page 931) value provides, in milliseconds, the amount of time spent remapping copy-on-write memory mapped views. Smaller values indicate better journal performance.

recordStats For an example of the `recordStats` output, see the *recordStats section* of the `http://docs.mongodb.org/manualreference/server-status` page.

serverStatus.**recordStats**

The `recordStats` (page 931) document provides fine grained reporting on page faults on a per database level.

MongoDB uses a read lock on each database to return `recordStats` (page 931). To minimize this overhead, you can disable this section, as in the following operation:

```
db.serverStatus( { recordStats: 0 } )
```

serverStatus.recordStats.**accessesNotInMemory**

`accessesNotInMemory` (page 931) reflects the number of times `mongod` (page 1049) needed to access a memory page that was *not* resident in memory for *all* databases managed by this `mongod` (page 1049) instance.

serverStatus.recordStats.**pageFaultExceptionsThrown**

`pageFaultExceptionsThrown` (page 931) reflects the number of page fault exceptions thrown by `mongod` (page 1049) when accessing data for *all* databases managed by this `mongod` (page 1049) instance.

serverStatus.recordStats.local.**accessesNotInMemory**

`accessesNotInMemory` (page 931) reflects the number of times `mongod` (page 1049) needed to access a memory page that was *not* resident in memory for the `local` database.

serverStatus.recordStats.local.**pageFaultExceptionsThrown**

`pageFaultExceptionsThrown` (page 931) reflects the number of page fault exceptions thrown by `mongod` (page 1049) when accessing data for the `local` database.

serverStatus.recordStats.admin.**accessesNotInMemory**

`accessesNotInMemory` (page 931) reflects the number of times `mongod` (page 1049) needed to access a memory page that was *not* resident in memory for the *admin database*.

serverStatus.recordStats.admin.**pageFaultExceptionsThrown**

`pageFaultExceptionsThrown` (page 931) reflects the number of page fault exceptions thrown by `mongod` (page 1049) when accessing data for the *admin database*.

serverStatus.recordStats.<database>.**accessesNotInMemory**

`accessesNotInMemory` (page 931) reflects the number of times `mongod` (page 1049) needed to access a memory page that was *not* resident in memory for the `<database>` database.

serverStatus.recordStats.<database>.**pageFaultExceptionsThrown**

`pageFaultExceptionsThrown` (page 931) reflects the number of page fault exceptions thrown by `mongod` (page 1049) when accessing data for the `<database>` database.

workingSet New in version 2.4.

Note: The `workingSet` (page 932) data is only included in the output of `serverStatus` (page 919) if explicitly enabled. To return the `workingSet` (page 932), use one of the following commands:

```
db.serverStatus( { workingSet: 1 } )
db.runCommand( { serverStatus: 1, workingSet: 1 } )
```

For an example of the workingSet output, see the *workingSet section* of the http://docs.mongodb.org/manualreference/server-status page.

serverStatus.**workingSet**

workingSet (page 932) is a document that contains values useful for estimating the size of the working set, which is the amount of data that MongoDB uses actively. workingSet (page 932) uses an internal data structure that tracks pages accessed by mongod (page 1049).

serverStatus.workingSet.**note**

note (page 932) is a field that holds a string warning that the workingSet (page 932) document is an estimate.

serverStatus.workingSet.**pagesInMemory**

pagesInMemory (page 932) contains a count of the total number of pages accessed by mongod (page 1049) over the period displayed in overSeconds (page 932). The default page size is 4 kilobytes: to convert this value to the amount of data in memory multiply this value by 4 kilobytes.

If your total working set is less than the size of physical memory, over time the value of pagesInMemory (page 932) will reflect your data size.

Use pagesInMemory (page 932) in conjunction with overSeconds (page 932) to help estimate the actual size of the working set.

serverStatus.workingSet.**computationTimeMicros**

computationTimeMicros (page 932) reports the amount of time the mongod (page 1049) instance used to compute the other fields in the workingSet (page 932) section.

Reporting on workingSet (page 932) may impact the performance of other operations on the mongod (page 1049) instance because MongoDB must collect some data within the context of a lock. Ensure that automated monitoring tools consider this metric when determining the frequency of collection for workingSet (page 932).

serverStatus.workingSet.**overSeconds**

overSeconds (page 932) returns the amount of time elapsed between the newest and oldest pages tracked in the pagesInMemory (page 932) data point.

If overSeconds (page 932) is decreasing, or if pagesInMemory (page 932) equals physical RAM *and* overSeconds (page 932) is very small, the working set may be much *larger* than physical RAM.

When overSeconds (page 932) is large, MongoDB's data set is equal to or *smaller* than physical RAM.

metrics For an example of the metrics output, see the *metrics section* of the http://docs.mongodb.org/manualreference/server-status page.

New in version 2.4.

serverStatus.**metrics**

The metrics (page 932) document holds a number of statistics that reflect the current use and state of a running mongod (page 1049) instance.

serverStatus.metrics.**document**

The document (page 932) holds a document of that reflect document access and modification patterns and data use. Compare these values to the data in the opcounters (page 928) document, which track total number of operations.

`serverStatus.metrics.document.`**`deleted`**
`deleted` (page 932) reports the total number of documents deleted.

`serverStatus.metrics.document.`**`inserted`**
`inserted` (page 933) reports the total number of documents inserted.

`serverStatus.metrics.document.`**`returned`**
`returned` (page 933) reports the total number of documents returned by queries.

`serverStatus.metrics.document.`**`updated`**
`updated` (page 933) reports the total number of documents updated.

`serverStatus.metrics.`**`getLastError`**
`getLastError` (page 933) is a document that reports on `getLastError` (page 861) use.

`serverStatus.metrics.getLastError.`**`wtime`**
`wtime` (page 933) is a sub-document that reports `getLastError` (page 861) operation counts with a `w` argument greater than `1`.

`serverStatus.metrics.getLastError.wtime.`**`num`**
`num` (page 933) reports the total number of `getLastError` (page 861) operations without a specified write concern (i.e. `w`) that wait for one or more members of a replica set to acknowledge the write operation (i.e. greater than `1`.)

`serverStatus.metrics.getLastError.wtime.`**`totalMillis`**
`totalMillis` (page 933) reports the total amount of time in milliseconds that the `mongod` (page 1049) has spent performing `getLastError` (page 861) operations with write concern (i.e. `w`) that wait for one or more members of a replica set to acknowledge the write operation (i.e. greater than `1`.)

`serverStatus.metrics.getLastError.`**`wtimeouts`**
`wtimeouts` (page 933) reports the number of times that *write concern* operations have timed out as a result of the `wtimeout` threshold to `getLastError` (page 861).

`serverStatus.metrics.`**`operation`**
`operation` (page 933) is a sub-document that holds counters for several types of update and query operations that MongoDB handles using special operation types.

`serverStatus.metrics.operation.`**`fastmod`**
`fastmod` (page 933) reports the number of *update* (page 93) operations that neither cause documents to grow nor require updates to the index. For example, this counter would record an update operation that use the `$inc` (page 810) operator to increment the value of a field that is not indexed.

`serverStatus.metrics.operation.`**`idhack`**
`idhack` (page 933) reports the number of queries that contain the `_id` field. For these queries, MongoDB will use default index on the `_id` field and skip all query plan analysis.

`serverStatus.metrics.operation.`**`scanAndOrder`**
`scanAndOrder` (page 933) reports the total number of queries that return sorted numbers that cannot perform the sort operation using an index.

`serverStatus.metrics.`**`queryExecutor`**
`queryExecutor` (page 933) is a document that reports data from the query execution system.

`serverStatus.metrics.queryExecutor.`**`scanned`**
`scanned` (page 933) reports the total number of index items scanned during queries and query-plan evaluation. This counter is the same as `nscanned` (page 982) in the output of `explain()` (page 979).

`serverStatus.metrics.`**`record`**
`record` (page 933) is a document that reports data related to record allocation in the on-disk memory files.

`serverStatus.metrics.record.`**`moves`**
`moves` (page 933) reports the total number of times documents move within the on-disk representation of the

MongoDB data set. Documents move as a result of operations that increase the size of the document beyond their allocated record size.

`serverStatus.metrics.repl`
: `repl` (page 934) holds a sub-document that reports metrics related to the replication process. `repl` (page 934) document appears on all `mongod` (page 1049) instances, even those that aren't members of *replica sets*.

`serverStatus.metrics.repl.apply`
: `apply` (page 934) holds a sub-document that reports on the application of operations from the replication *oplog*.

`serverStatus.metrics.repl.apply.batches`
: `batches` (page 934) reports on the oplog application process on *secondaries* members of replica sets. See *Multithreaded Replication* (page 407) for more information on the oplog application processes

`serverStatus.metrics.repl.apply.batches.num`
: `num` (page 934) reports the total number of batches applied across all databases.

`serverStatus.metrics.repl.apply.batches.totalMillis`
: `totalMillis` (page 934) reports the total amount of time the `mongod` (page 1049) has spent applying operations from the oplog.

`serverStatus.metrics.repl.apply.ops`
: `ops` (page 934) reports the total number of *oplog* operations applied.

`serverStatus.metrics.repl.buffer`
: MongoDB buffers oplog operations from the replication sync source buffer before applying oplog entries in a batch. `buffer` (page 934) provides a way to track the oplog buffer. See *Multithreaded Replication* (page 407) for more information on the oplog application process.

`serverStatus.metrics.repl.buffer.count`
: `count` (page 934) reports the current number of operations in the oplog buffer.

`serverStatus.metrics.repl.buffer.maxSizeBytes`
: `maxSizeBytes` (page 934) reports the maximum size of the buffer. This value is a constant setting in the `mongod` (page 1049), and is not configurable.

`serverStatus.metrics.repl.buffer.sizeBytes`
: `sizeBytes` (page 934) reports the current size of the contents of the oplog buffer.

`serverStatus.metrics.repl.network`
: `network` (page 934) reports network use by the replication process.

`serverStatus.metrics.repl.network.bytes`
: `bytes` (page 934) reports the total amount of data read from the replication sync source.

`serverStatus.metrics.repl.network.getmores`
: `getmores` (page 934) reports on the `getmore` operations, which are requests for additional results from the oplog *cursor* as part of the oplog replication process.

`serverStatus.metrics.repl.network.getmores.num`
: `num` (page 934) reports the total number of `getmore` operations, which are operations that request an additional set of operations from the replication sync source.

`serverStatus.metrics.repl.network.getmores.totalMillis`
: `totalMillis` (page 934) reports the total amount of time required to collect data from `getmore` operations.

Note: This number can be quite large, as MongoDB will wait for more data even if the `getmore` operation does not initial return data.

`serverStatus.metrics.repl.network.ops`
: `ops` (page 934) reports the total number of operations read from the replication source.

serverStatus.metrics.repl.network.**readersCreated**

readersCreated (page 934) reports the total number of oplog query processes created. MongoDB will create a new oplog query any time an error occurs in the connection, including a timeout, or a network operation. Furthermore, readersCreated (page 934) will increment every time MongoDB selects a new source fore replication.

serverStatus.metrics.repl.**oplog**

oplog (page 935) is a document that reports on the size and use of the *oplog* by this mongod (page 1049) instance.

serverStatus.metrics.repl.oplog.**insert**

insert (page 935) is a document that reports insert operations into the *oplog*.

serverStatus.metrics.repl.oplog.insert.**num**

num (page 935) reports the total number of items inserted into the *oplog*.

serverStatus.metrics.repl.oplog.insert.**totalMillis**

totalMillis (page 935) reports the total amount of time spent for the mongod (page 1049) to insert data into the *oplog*.

serverStatus.metrics.repl.oplog.**insertBytes**

insertBytes (page 935) the total size of documents inserted into the oplog.

serverStatus.metrics.repl.**preload**

preload (page 935) reports on the "pre-fetch" stage, where MongoDB loads documents and indexes into RAM to improve replication throughput.

See *Multithreaded Replication* (page 407) for more information about the *pre-fetch* stage of the replication process.

serverStatus.metrics.repl.preload.**docs**

docs (page 935) is a sub-document that reports on the documents loaded into memory during the *pre-fetch* stage.

serverStatus.metrics.repl.preload.docs.**num**

num (page 935) reports the total number of documents loaded during the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.docs.**totalMillis**

totalMillis (page 935) reports the total amount of time spent loading documents as part of the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.**indexes**

indexes (page 935) is a sub-document that reports on the index items loaded into memory during the *pre-fetch* stage of replication.

See *Multithreaded Replication* (page 407) for more information about the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.indexes.**num**

num (page 935) reports the total number of index entries loaded by members before updating documents as part of the *pre-fetch* stage of replication.

serverStatus.metrics.repl.preload.indexes.**totalMillis**

totalMillis (page 935) reports the total amount of time spent loading index entries as part of the *pre-fetch* stage of replication.

serverStatus.metrics.**ttl**

ttl (page 935) is a sub-document that reports on the operation of the resource use of the *ttl index* (page 599) process.

serverStatus.metrics.ttl.**deletedDocuments**

deletedDocuments (page 935) reports the total number of documents deleted from collections with a *ttl index* (page 599).

serverStatus.metrics.ttl.passes

passes (page 935) reports the number of times the background process removes documents from collections with a *ttl index* (page 599).

features

features

features (page 936) is an internal command that returns the build-level feature settings.

isSelf

_isSelf

_isSelf (page 936) is an internal command.

55.2.3 Internal Commands

Internal Commands

Name	Description
handshake (page 936)	Internal command.
_recvChunkAbort (page 936)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkCommit (page 937)	Internal command that supports chunk migrations in sharded clusters. Do not call directly.
_recvChunkStart (page 937)	Internal command that facilitates chunk migrations in sharded clusters.. Do not call directly.
_recvChunkStatus (page 937)	Internal command that returns data to support chunk migrations in sharded clusters. Do not call directly.
_replSetFresh	Internal command that supports replica set election operations.
mapreduce.shardedfinish (page 937)	Internal command that supports *map-reduce* in *sharded cluster* environments.
_transferMods (page 937)	Internal command that supports chunk migrations. Do not call directly.
replSetHeartbeat (page 937)	Internal command that supports replica set operations.
replSetGetRBID (page 937)	Internal command that supports replica set operations.
_migrateClone (page 938)	Internal command that supports chunk migration. Do not call directly.
replSetElect (page 938)	Internal command that supports replica set functionality.
writeBacksQueued (page 938)	Internal command that supports chunk migrations in sharded clusters.
writebacklisten (page 939)	Internal command that supports chunk migrations in sharded clusters.

handshake

handshake

handshake (page 936) is an internal command.

recvChunkAbort

_recvChunkAbort

_recvChunkAbort (page 936) is an internal command. Do not call directly.

recvChunkCommit

_recvChunkCommit
_recvChunkCommit (page 937) is an internal command. Do not call directly.

recvChunkStart

_recvChunkStart
_recvChunkStart (page 937) 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

_recvChunkStatus
_recvChunkStatus (page 937) is an internal command. Do not call directly.

replSetFresh

replSetFresh
replSetFresh (page 937) is an internal command that supports replica set functionality.

mapreduce.shardedfinish

mapreduce.**shardedfinish**
Provides internal functionality to support *map-reduce* in *sharded* environments.

See also:

"mapReduce (page 840)"

transferMods

_transferMods
_transferMods (page 937) is an internal command. Do not call directly.

replSetHeartbeat

replSetHeartbeat
replSetHeartbeat (page 937) is an internal command that supports replica set functionality.

replSetGetRBID

replSetGetRBID
replSetGetRBID (page 937) is an internal command that supports replica set functionality.

migrateClone

_migrateClone

_migrateClone (page 938) is an internal command. Do not call directly.

replSetElect

replSetElect

replSetElect (page 938) is an internal command that support replica set functionality.

writeBacksQueued

writeBacksQueued

writeBacksQueued (page 938) is an internal command that returns a document reporting there are operations in the write back queue for the given mongos (page 1061) and information about the queues.

writeBacksQueued.**hasOpsQueued**

Boolean.

hasOpsQueued (page 938) is `true` if there are `write Back` operations queued.

writeBacksQueued.**totalOpsQueued**

Integer.

totalOpsQueued (page 938) reflects the number of operations queued.

writeBacksQueued.**queues**

Document.

queues (page 938) holds a sub-document where the fields are all write back queues. These field hold a document with two fields that reports on the state of the queue. The fields in these documents are:

writeBacksQueued.queues.**n**

n (page 938) reflects the size, by number of items, in the queues.

writeBacksQueued.queues.**minutesSinceLastCall**

The number of minutes since the last time the mongos (page 1061) touched this queue.

The command document has the following prototype form:

```
{writeBacksQueued: 1}
```

To call writeBacksQueued (page 938) from the mongo (page 1066) shell, use the following db.runCommand() (page 1012) form:

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

Consider the following example output:

```
{
  "hasOpsQueued" : true,
  "totalOpsQueued" : 7,
  "queues" : {
          "50b4f09f6671b11ff1944089" : { "n" : 0, "minutesSinceLastCall" : 1 },
          "50b4f09fc332bf1c5aeaaf59" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4f09f6671b1d51df98cb6" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4f0c67ccf1e5c6effb72e" : { "n" : 0, "minutesSinceLastCall" : 0 },
          "50b4faf12319f193cfdec0d1" : { "n" : 0, "minutesSinceLastCall" : 4 },
```

```
            "50b4f013d2c1f8d62453017e" : { "n" : 0, "minutesSinceLastCall" : 0 },
            "50b4f0f12319f193cfdec0d1" : { "n" : 0, "minutesSinceLastCall" : 1 }
    },
    "ok" : 1
}
```

writebacklisten

writebacklisten

writebacklisten (page 939) is an internal command.

55.2.4 Testing Commands

Testing Commands

Name	Description
testDistLockWithSkew	Internal command. Do not call this directly.
testDistLockWithSyncCluster	Internal command. Do not call this directly.
captrunc (page 940)	Internal command. Truncates capped collections.
emptycapped (page 940)	Internal command. Removes all documents from a capped collection.
godinsert (page 941)	Internal command for testing.
_hashBSONElement (page 941)	Internal command. Computes the MD5 hash of a BSON element.
_journalLatencyTest	Tests the time required to write and perform a file system sync for a file in the journal directory.
sleep (page 942)	Internal command for testing. Forces MongoDB to block all operations.
replSetTest (page 943)	Internal command for testing replica set functionality.
forceerror (page 943)	Internal command for testing. Forces a user assertion exception.
skewClockCommand	Internal command. Do not call this command directly.
configureFailPoint (page 943)	Internal command for testing. Configures failure points.

testDistLockWithSkew

_testDistLockWithSkew

_testDistLockWithSkew (page 939) is an internal command. Do not call directly.

Note: _testDistLockWithSkew (page 939) is an internal command that is not enabled by default. _testDistLockWithSkew (page 939) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. _testDistLockWithSkew (page 939) cannot be enabled during run-time.

testDistLockWithSyncCluster

_testDistLockWithSyncCluster

_testDistLockWithSyncCluster (page 939) is an internal command. Do not call directly.

Note: _testDistLockWithSyncCluster (page 939) is an internal command that is not enabled by default. `_testDistLockWithSyncCluster` (page 939) must be enabled by using *--setParameter enableTestCommands=1* on the `mongod` (page 1049) command line. `_testDistLockWithSyncCluster` (page 939) cannot be enabled during run-time.

captrunc

Definition

captrunc

Truncates capped collections. `captrunc` (page 940) is an internal command to support testing that takes the following form:

```
{ captrunc: "<collection>", n: <integer>, inc: <true|false> }.
```

`captrunc` (page 940) has the following fields:

field string captrunc The name of the collection to truncate.

field integer n The number of documents to remove from the collection.

field boolean inc Specifies whether to truncate the nth document.

Note: `captrunc` (page 940) is an internal command that is not enabled by default. `captrunc` (page 940) must be enabled by using *--setParameter enableTestCommands=1* on the `mongod` (page 1049) command line. `captrunc` (page 940) cannot be enabled during run-time.

Examples The following command truncates 10 older documents from the collection `records`:

```
db.runCommand({captrunc: "records" , n: 10})
```

The following command truncates 100 documents and the 101st document:

```
db.runCommand({captrunc: "records", n: 100, inc: true})
```

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.

Note: `emptycapped` (page 940) is an internal command that is not enabled by default. `emptycapped` (page 940) must be enabled by using *--setParameter enableTestCommands=1* on the `mongod` (page 1049) command line. `emptycapped` (page 940) cannot be enabled during run-time.

godinsert

godinsert

godinsert (page 941) 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.

Note: godinsert (page 941) is an internal command that is not enabled by default. godinsert (page 941) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. godinsert (page 941) cannot be enabled during run-time.

_hashBSONElement

Description

_hashBSONElement

New in version 2.4.

An internal command that computes the MD5 hash of a BSON element. The _hashBSONElement (page 941) command returns 8 bytes from the 16 byte MD5 hash.

The _hashBSONElement (page 941) command has the following fields:

field BSONElement key The BSON element to hash.

field integer seed A seed used when computing the hash.

Note: _hashBSONElement (page 941) is an internal command that is not enabled by default. _hashBSONElement (page 941) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. _hashBSONElement (page 941) cannot be enabled during run-time.

Output The _hashBSONElement (page 941) command returns a document that holds the following fields:

_hashBSONElement.**key**

The original BSON element.

_hashBSONElement.**seed**

The seed used for the hash, defaults to `0`.

_hashBSONElement.**out**

The decimal result of the hash.

_hashBSONElement.**ok**

Holds the `1` if the function returns successfully, and `0` if the operation encountered an error.

Example Invoke a mongod (page 1049) instance with test commands enabled:

```
mongod --setParameter enableTestCommands=1
```

Run the following to compute the hash of an ISODate string:

```
db.runCommand({_hashBSONElement: ISODate("2013-02-12T22:12:57.211Z")})
```

The command returns the following document:

```
{
  "key" : ISODate("2013-02-12T22:12:57.211Z"),
  "seed" : 0,
  "out" : NumberLong("-4185544074338741873"),
  "ok" : 1
}
```

Run the following to hash the same ISODate string but this time to specify a seed value:

```
db.runCommand({_hashBSONElement: ISODate("2013-02-12T22:12:57.211Z"), seed:2013})
```

The command returns the following document:

```
{
  "key" : ISODate("2013-02-12T22:12:57.211Z"),
  "seed" : 2013,
  "out" : NumberLong("7845924651247493302"),
  "ok" : 1
}
```

journalLatencyTest

journalLatencyTest

`journalLatencyTest` (page 942) 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 942) command against the *admin database* in the form:

```
{ journalLatencyTest: 1 }
```

The value (i.e. `1` above), does not affect the operation of the command.

Note: `journalLatencyTest` (page 942) is an internal command that is not enabled by default. `journalLatencyTest` (page 942) must be enabled by using `--setParameter enableTestCommands=1` on the `mongod` (page 1049) command line. `journalLatencyTest` (page 942) cannot be enabled during run-time.

sleep

Definition

sleep

Forces the database to block all operations. This is an internal command for testing purposes.

The `sleep` (page 942) command takes a document with the following fields:

field boolean w If true, obtains a global write lock. Otherwise obtains a read lock.

field integer secs The number of seconds to sleep.

Example

The `sleep` (page 942) command takes the following prototype form:

```
{ sleep: { w: <true|false>, secs: <seconds> } }
```

The command places the mongod (page 1049) instance in a *write lock* state for a specified number of seconds. Without arguments, sleep (page 942) causes a "read lock" for 100 seconds.

Warning: sleep (page 942) claims the lock specified in the w argument and blocks *all* operations on the mongod (page 1049) instance for the specified amount of time.

Note: sleep (page 942) is an internal command that is not enabled by default. sleep (page 942) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. sleep (page 942) cannot be enabled during run-time.

replSetTest

replSetTest

replSetTest (page 943) is internal diagnostic command used for regression tests that supports replica set functionality.

Note: replSetTest (page 943) is an internal command that is not enabled by default. replSetTest (page 943) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. replSetTest (page 943) cannot be enabled during run-time.

forceerror

forceerror

The forceerror (page 943) command is for testing purposes only. Use forceerror (page 943) to force a user assertion exception. This command always returns an ok value of 0.

skewClockCommand

_skewClockCommand

_skewClockCommand (page 943) is an internal command. Do not call directly.

Note: _skewClockCommand (page 943) is an internal command that is not enabled by default. _skewClockCommand (page 943) must be enabled by using *--setParameter enableTestCommands=1* on the mongod (page 1049) command line. _skewClockCommand (page 943) cannot be enabled during run-time.

configureFailPoint

Definition

configureFailPoint

Configures a failure point that you can turn on and off while MongoDB runs. configureFailPoint (page 943) is an internal command for testing purposes that takes the following form:

```
{configureFailPoint: "<failure_point>", mode: <behavior> }
```

You must issue configureFailPoint (page 943) against the *admin database*. configureFailPoint (page 943) has the following fields:

field string configureFailPoint The name of the failure point.

field document,string mode Controls the behavior of a failure point. The possible values are `alwaysOn`, `off`, or a document in the form of `{times: n}` that specifies the number of times the failure point remains on before it deactivates. The maximum value for the number is a 32-bit signed integer.

field document data Passes in additional data for use in configuring the fail point. For example, to imitate a slow connection pass in a document that contains a delay time.

Note: `configureFailPoint` (page 943) is an internal command that is not enabled by default. `configureFailPoint` (page 943) must be enabled by using `--setParameter enableTestCommands=1` on the `mongod` (page 1049) command line. `configureFailPoint` (page 943) cannot be enabled during run-time.

Example

```
db.adminCommand( { configureFailPoint: "blocking_thread", mode: {times: 21} } )
```

55.3 `mongo` Shell Methods

JavaScript in MongoDB

Although these methods use JavaScript, most interactions with MongoDB do not use JavaScript but use an *idiomatic driver* (page 575) in the language of the interacting application.

55.3.1 Collection

Collection Methods

Name	Description
`db.collection.aggregate()` (page 945)	Provides access to the *aggregation framework* (page 247) *pipelin*
`db.collection.count()` (page 946)	Wraps `count` (page 834) to return a count of the number of docu
`db.collection.createIndex()` (page 946)	Builds an index on a collection. Use `db.collection.ensur`
`db.collection.getIndexStats()` (page 947)	Renders a human-readable view of the data collected by `indexS`
`db.collection.dataSize()` (page 948)	Returns the size of the collection. Wraps the `size` (page 901) fie
`db.collection.distinct()` (page 948)	Returns an array of documents that have distinct values for the sp
`db.collection.drop()` (page 948)	Removes the specified collection from the database.
`db.collection.dropIndex()`	Removes a specified index on a collection.
`db.collection.dropIndexes()` (page 949)	Removes all indexes on a collection.
`db.collection.ensureIndex()` (page 949)	Creates an index if it does not currently exist. If the index exists e

Table 55.1 – continued from p

Name	Description
db.collection.find() (page 951)	Performs a query on a collection and returns a cursor object.
db.collection.findAndModify() (page 952)	Atomically modifies and returns a single document.
db.collection.findOne() (page 955)	Performs a query and returns a single document.
db.collection.getIndexes() (page 956)	Returns an array of documents that describe the existing indexes c
db.collection.getShardDistribution() (page 956)	For collections in sharded clusters, db.collection.getSha
db.collection.getShardVersion() (page 958)	Internal diagnostic method for shard cluster.
db.collection.group() (page 958)	Provides simple data aggregation function. Groups documents in
db.collection.insert() (page 961)	Creates a new document in a collection.
db.collection.isCapped() (page 963)	Reports if a collection is a *capped collection*.
db.collection.mapReduce() (page 963)	Performs map-reduce style data aggregation.
db.collection.reIndex() (page 970)	Rebuilds all existing indexes on a collection.
db.collection.remove() (page 970)	Deletes documents from a collection.
db.collection.renameCollection() (page 971)	Changes the name of a collection.
db.collection.save() (page 972)	Provides a wrapper around an insert() (page 961) and updat
db.collection.stats() (page 973)	Reports on the state of a collection. Provides a wrapper around th
db.collection.storageSize() (page 973)	Reports the total size used by the collection. Provides a wrapper a
db.collection.totalSize() (page 973)	Reports the total size of a collection, including the size of all docu
db.collection.totalIndexSize() (page 973)	Reports the total size used by the indexes on a collection. Provide
db.collection.update() (page 974)	Modifies a document in a collection.
db.collection.validate() (page 976)	Performs diagnostic operations on a collection.

db.collection.aggregate()

Definition

`db.collection.aggregate(pipeline)`

New in version 2.2.

Calculates aggregate values for the data in a collection. Always call the `aggregate()` (page 945) method on a collection object.

param document pipeline A sequence of data aggregation processes. See the *aggregation reference* (page 263) for documentation of these operators.

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.

Throws exception

Changed in version 2.4: If an error occurs, the `aggregate()` (page 945) helper throws an exception. In previous versions, the helper returned a document with the error message and code, and `ok` status field not equal to `1`, same as the `aggregate` (page 834) command.

Example The following is an example from the *aggregation documentation* (page 247).

```
db.article.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
```

```
  { $unwind : "$tags" },
  { $group : {
     _id : { tags : 1 },
     authors : { $addToSet : "$author" }
  } }
);
```

See also:

"aggregate (page 834)," "*Aggregation Framework* (page 247)," and "*Aggregation Framework Reference* (page 263)."

db.collection.count()

Definition

`db.collection.count(<query>)`

Returns the count of documents that would match a `find()` (page 951) query. The `db.collection.count()` (page 946) method does not perform the `find()` (page 951) operation but instead counts and returns the number of results that match a query.

The `db.collection.count()` (page 946) method has the following parameter:

param document query The query selection criteria.

See also:

`cursor.count()` (page 978)

Examples

Count all Documents in a Collection To count the number of all documents in the `orders` collection, use the following operation:

```
db.orders.count()
```

This operation is equivalent to the following:

```
db.orders.find().count()
```

Count all Documents that Match a Query Count the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')`:

```
db.orders.count( { ord_dt: { $gt: new Date('01/01/2012') } } )
```

The query is equivalent to the following:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).count()
```

db.collection.createIndex()

Definition

`db.collection.createIndex(keys, options)`

Deprecated since version 1.8.

Creates indexes on collections.

param document keys For each field to index, a key-value pair with the field and the index order: `1` for ascending or `-1` for descending.

param document options One or more key-value pairs that specify index options. For a list of options, see `db.collection.ensureIndex()` (page 949).

See also:

Indexes (page 307), `db.collection.createIndex()` (page 946), `db.collection.dropIndex()`, `db.collection.dropIndexes()` (page 949), `db.collection.getIndexes()` (page 956), `db.collection.reIndex()` (page 970), and `db.collection.totalIndexSize()` (page 973)

db.collection.getIndexStats()

Definition

`db.collection.getIndexStats(index)`

Displays a human-readable summary of aggregated statistics about an index's B-tree data structure. The information summarizes the output returned by the `indexStats` (page 911) command and `indexStats()` method. The `getIndexStats` (page 947) method displays the information on the screen and does not return an object.

The `getIndexStats()` (page 947) method has the following form:

```
db.<collection>.getIndexStats( { index : "<index name>" } )
```

param document index The *index name* (page 315).

The `getIndexStats()` (page 947) method is available only when connected to a `mongod` (page 1049) instance that uses the `--enableExperimentalIndexStatsCmd` option.

To view *index names* (page 315) for a collection, use the `getIndexes()` (page 956) method.

> **Warning:** Do not use `getIndexStats()` (page 947) or `indexStats` (page 911) with production deployments.

Example The following command returns information for an index named `type_1_traits_1`:

```
db.animals.getIndexStats({index:"type_1_traits_1"})
```

The command returns the following summary. For more information on the B-tree statistics, see `indexStats` (page 911).

```
-- index "undefined" --
  version 1 | key pattern {  "type" : 1,  "traits" : 1 } | storage namespace "test.animals.$type_1_t
  2 deep, bucket body is 8154 bytes

  bucket count       45513   on average 99.401 % (±0.463 %) full     49.581 % (±4.135 %) bson keys,

  -- depth 0 --
    bucket count     1       on average 71.511 % (±0.000 %) full     36.191 % (±0.000 %) bson keys,

  -- depth 1 --
    bucket count     180     on average 98.954 % (±5.874 %) full     49.732 % (±5.072 %) bson keys,

  -- depth 2 --
    bucket count     45332   on average 99.403 % (±0.245 %) full     49.580 % (±4.130 %) bson keys,
```

db.collection.dataSize()

`db.collection.dataSize()`

Returns The size of the collection. This method provides a wrapper around the `size` (page 901) output of the `collStats` (page 900) (i.e. `db.collection.stats()` (page 973)) command.

db.collection.distinct()

Definition

`db.collection.distinct(field, query)`

Finds the distinct values for a specified field across a single collection and returns the results in an array.

param string field The field for which to return distinct values.

param document query A query that specifies the documents from which to retrieve the distinct values.

The `db.collection.distinct()` (page 948) method provides a wrapper around the `distinct` (page 835) command. Results must not be larger than the maximum *BSON size* (page 1139).

When possible to use covered indexes, the `db.collection.distinct()` (page 948) method will use an index to find the documents in the query as well as to return the data.

Examples The following are examples of the `db.collection.distinct()` (page 948) method:

- Return an array of the distinct values of the field `ord_dt` from all documents in the `orders` collection:

```
db.orders.distinct( 'ord_dt' )
```

- Return an array of the distinct values of the field `sku` in the subdocument `item` from all documents in the `orders` collection:

```
db.orders.distinct( 'item.sku' )
```

- Return an array of the distinct values of the field `ord_dt` from the documents in the `orders` collection where the `price` is greater than `10`:

```
db.orders.distinct( 'ord_dt', { price: { $gt: 10 } } )
```

db.collection.drop()

`db.collection.drop()`

Call the `db.collection.drop()` (page 948) method on a collection to drop it from the database.

`db.collection.drop()` (page 948) takes no arguments and will produce an error if called with any arguments.

db.collection.dropIndexes()

`db.collection.dropIndexes()`

Drops all indexes other than the required index on the `_id` field. Only call `dropIndexes()` (page 949) as a method on a collection object.

db.collection.dropIndexes()

`db.collection.dropIndexes()`

Drops all indexes other than the required index on the `_id` field. Only call `dropIndexes()` (page 949) as a method on a collection object.

db.collection.ensureIndex()

Definition

`db.collection.ensureIndex(keys, options)`

Creates an index on the specified field if the index does not already exist.

The `db.collection.ensureIndex()` (page 949) method has the following fields:

param document keys For ascending/descending indexes, a *document* (page 66) 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. MongoDB supports several different index types including *text* (page 318), *geospatial* (page 339), and *hashed* (page 315) indexes.

param document options A document that controls the creation of the index. The document contains a set of options, as described in the next table.

> **Warning:** Index names, including their full namespace (i.e. `database.collection`) cannot be longer than 128 characters. See the `db.collection.getIndexes()` (page 956) field `name` (page 956) for the names of existing indexes.

The `options` document has one or more of the following fields:

param Boolean background Builds the index in the background so that building an index does *not* block other database activities. Specify `true` to build in the background. The default value is `false`.

param Boolean unique Creates a unique index so that the collection will not accept insertion of documents where the index key or keys match an existing value in the index. Specify `true` to create a unique index. The default value is `false`. This option applies only to ascending/descending indexes.

param string name The name of the index. If unspecified, MongoDB generates an index name by concatenating the names of the indexed fields and the sort order.

param Boolean dropDups Creates a unique index on a field that *may* have duplicates. MongoDB indexes only the first occurrence of a key and **removes** all documents from the collection that contain subsequent occurrences of that key. Specify `true` to create unique index. The default value is `false`. This option applies only to scalar indexes.

param Boolean sparse If `true`, the index only references documents with the specified field. These indexes use less space but behave differently in some situations (particularly sorts). The default value is `false`. This applies only to ascending/descending indexes.

param integer expireAfterSeconds Specifies a value, in seconds, as a *TTL* to control how long MongoDB retains documents in this collection. See *Expire Data from Collections by Setting TTL* (page 599) for more information on this functionality. This applies only to *TTL* indexes.

param index version v The index version number. The default index version depends on the version of `mongod` (page 1049) running when creating the index. Before version 2.0, the this value was 0; versions 2.0 and later use version 1, which provides a smaller and faster index format. Specify a different index version *only* in unusual situations.

param document weights For `text` indexes, the significance of the field relative to the other indexed fields. The document contains field and weight pairs. The weight is a number ranging from 1 to 99,999 and denotes the significance of the field relative to the other indexed fields in terms of the score. You can specify weights for some or all the indexed fields. See *Control Results of Text Search with Weights* (page 357) to adjust the scores. The default value is `1`. This applies to `text` indexes only.

param string default_language For a `text` index, the language that determines the list of stop words and the rules for the stemmer and tokenizer. See *Text Search Languages* (page 860) for the available languages and *Specify a Language for Text Index* (page 355) for more information and examples. The default value is `english`. This applies to `text` indexes only.

param string language_override For a `text` index, specify the name of the field in the document that contains, for that document, the language to override the default language. See *Create a text Index on a Multi-language Collection* (page 356). The default value is `language`.

Examples

Create an Ascending Index on a Single Field The following example creates an ascending index on the field `orderDate`.

```
db.collection.ensureIndex( { orderDate: 1 } )
```

If the `keys` document specifies more than one field, then `db.collection.ensureIndex()` (page 949) creates a *compound index*.

Create an Index on a Multiple Fields The following example creates a compound index on the `orderDate` field (in ascending order) and the `zipcode` field (in descending order.)

```
db.collection.ensureIndex( { orderDate: 1, zipcode: -1 } )
```

A compound index cannot include a *hashed index* (page 315) component.

Note: The order of an index is important for supporting `cursor.sort()` (page 991) operations using the index.

See also:

- The *Indexes* (page 307) section of this manual for full documentation of indexes and indexing in MongoDB.
- The *Create a text Index* (page 354) section for more information and examples on creating `text` indexes.

Behaviors The `ensureIndex()` (page 949) method has the behaviors described here.

- To add or change index options you must drop the index using the `dropIndex()` method and issue another `ensureIndex()` (page 949) operation with the new options.

  If you create an index with one set of options, and then issue the `ensureIndex()` (page 949) method with the same index fields and different options without first dropping the index, `ensureIndex()` (page 949) will *not* rebuild the existing index with the new options.

- If you call multiple `ensureIndex()` (page 949) 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.

See also:

In addition to the ascending/descending indexes, MongoDB provides the following index types to provide additional functionalities:

- *TTL Indexes* (page 317) to support expiration of data,
- *Geospatial Indexes* (page 318) and *Geohaystack Indexes* (page 318) to support geospatial queries, and
- *text Indexes* (page 318) to support text searches.

db.collection.find()

Definition

`db.collection.find(<criteria>, <projection>)`

Selects documents in a collection and returns a *cursor* to the selected documents.

param document criteria Specifies selection criteria using *query operators* (page 785). To return all documents in a collection, omit this parameter or pass an empty document ({}).

param document projection Specifies the fields to return using *projection operators* (page 822). To return all fields in the matching document, omit this parameter.

Returns A *cursor* to the documents that match the `criteria` criteria. If the `projection` argument is specified, the matching documents contain only the `projection` fields, and the `_id` field if you do not explicitly exclude the `_id` field.

In the `mongo` (page 1066) shell, you can access the returned documents directly without explicitly using the JavaScript cursor handling method. Executing the query directly on the `mongo` (page 1066) shell prompt automatically iterates the cursor to display up to the first 20 documents. Type `it` to continue iteration.

Examples

Select All Documents in a Collection Call the `find()` (page 951) method with no parameters:

```
db.products.find()
```

This returns all the documents with all the fields from the collection `products`. In the `mongo` (page 1066) shell, the cursor returns the first batch of 20 matching documents by default. Iterate through the next batch by typing `it`. Use the appropriate cursor handling mechanism for your specific language driver.

Select Documents that Match a Selection Criteria Call the `find()` (page 951) method with a `criteria`:

```
db.products.find( { qty: { $gt: 25 } } )
```

This returns all the documents from the collection `products` where `qty` is greater than `25`, including all fields.

Prototype Projection Document

```
{ field1: boolean, field2: boolean ... }
```

The `boolean` can take the following include or exclude values:

- `1` or `true` to include. The `find()` (page 951) 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.

Project Certain Fields in the Result Set The following example selects documents that match a selection criteria and returns, or *projects*, only certain fields into the result set:

```
db.products.find( { qty: { $gt: 25 } }, { item: 1, qty: 1 } )
```

This 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()` (page 951) 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 }
```

Exclude Certain Fields from the Result Set The following example selects documents that match a selection criteria and excludes a set of fields from the resulting documents:

```
db.products.find( { qty: { $gt: 25 } }, { _id: 0, qty: 0 } )
```

The query returns 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()

Definition

`db.collection.`**`findAndModify`**`(`*<document>*`)`

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 `findAndModify()` (page 952) method is a shell helper around the `findAndModify` (page 851) command.

The `findAndModify()` (page 952) method has the following form:

```
db.collection.findAndModify( {
                                 query: <document>,
                                 sort: <document>,
                                 remove: <boolean>,
                                 update: <document>,
                                 new: <boolean>,
                                 fields: <document>,
                                 upsert: <boolean>
                             } );
```

The `db.collection.findAndModify()` (page 952) method takes a document parameter with the following subdocument fields:

param document query The selection criteria for the modification. The `query` field employs the same *query selectors* (page 785) as used in the `db.collection.find()` (page 951) method. Although the query may match multiple documents, `findAndModify()` (page 952) will select only one document to modify.

param document sort Determines which document the operation modifies if the query selects multiple documents. `findAndModify()` (page 952) modifies the first document in the sort order specified by this argument.

param Boolean remove Must specify either the `remove` or the `update` field in the `findAndModify()` (page 952) method. Removes the document specified in the `update` field. Set this to `true` to remove the selected document . The default is `false`.

param document update Must specify either the `remove` or the `update` field in the `findAndModify()` (page 952) method. Performs an update of the selected document. The `update` field employs the same *update operators* (page 810) or `field: value` specifications to modify the selected document.

param Boolean new When `true`, returns the modified document rather than the original. The `findAndModify()` (page 952) method ignores the `new` option for `remove` operations. The default is `false`.

param document fields A subset of fields to return. The `fields` document specifies an inclusion of a field with `1`, as in: `fields: { <field1>: 1, <field2>: 1, ... }`. See *projection* (page 45).

param Boolean upsert Used in conjunction with the `update` field. When `true`, `findAndModify()` (page 952) creates a new document if the `query` returns no documents. The default is `false`.

Return Data The `findAndModify()` (page 952) method returns either: the pre-modification document or, if `new: true` is set, the modified document.

Note:

- If the query finds no document for `update` or `remove` operations, `findAndModify()` (page 952) returns `null`.
- If the query finds no document for an `upsert`, operation, `findAndModify()` (page 952) performs an insert. If `new` is `false`, **and** the `sort` option is **NOT** specified, the method returns `null`.

  Changed in version 2.2: Previously returned an empty document `{}`. See *the 2.2 release notes* (page 1192) for more information.
- If the query finds no document for an `upsert`, `findAndModify()` (page 952) performs an insert. If `new` is `false`, **and** a `sort` option, the method returns an empty document `{}`.

Behaviors

Upsert and Unique Index When `findAndModify()` (page 952) includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients each invoke the method with the same `query` condition and these methods complete the `find` phase before any of methods perform the `modify` phase, these methods could insert the same document.

In the following example, no document with the name `Andy` exists, and multiple clients issue the following command:

```
db.people.findAndModify( {
    query: { name: "Andy" },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true
} )
```

Then, if these clients' `findAndModify()` (page 952) methods finish the `query` phase before any command starts the `modify` phase, **and** there is no unique index on the `name` field, the commands may all perform an upsert. To prevent this condition, create a *unique index* (page 314) on the `name` field. With the unique index in place, the multiple methods would observe one of the following behaviors:

- Exactly one `findAndModify()` (page 952) would successfully insert a new document.
- Zero or more `findAndModify()` (page 952) methods would update the newly inserted document.
- Zero or more `findAndModify()` (page 952) methods would fail when they attempted to insert a duplicate. If the method fails due to a unique index constraint violation, you can retry the method. Absent a delete of the document, the retry should not fail.

Sharded Collections When using `findAndModify` (page 851) in a *sharded* environment, the `query` **must** contain the *shard key* for all operations against the shard cluster for the *sharded* collections.

`findAndModify` (page 851) operations issued against `mongos` (page 1061) instances for *non-sharded* collections function normally.

Examples

Update The following method updates an existing document in the people collection where the document matches the query criteria:

```
db.people.findAndModify( {
    query: { name: "Tom", state: "active", rating: { $gt: 10 } },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } }
} )
```

This method 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` (page 786) 10.
2. The `sort` orders the results of the query in ascending order. If multiple documents meet the `query` condition, the method will select for modification the first document as ordered by this `sort`.
3. The update `increments` (page 810) the value of the `score` field by 1.
4. The method returns the original (i.e. pre-modification) document selected for this update:

   ```
   {
     "_id" : ObjectId("50f1e2c99beb36a0f45c6453"),
     "name" : "Tom",
     "state" : "active",
     "rating" : 100,
     "score" : 5
   }
   ```

   To return the modified document, add the `new:true` option to the method.

   If no document matched the `query` condition, the method returns `null`:

   ```
   null
   ```

Update and Insert The following method includes the `upsert: true` option to insert a new document if no document matches the `query` condition:

```
db.people.findAndModify( {
    query: { name: "Gus", state: "active", rating: 100 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true
  } )
```

If the method does **not** find a matching document, the method performs an upsert. Because the method included the `sort` option, it returns an empty document `{ }` as the original (pre-modification) document:

```
{ }
```

If the method did **not** include a `sort` option, the method returns `null`.

```
null
```

Update, Insert and Return New Document The following method includes both the `upsert: true` option and the `new:true` option to return the newly inserted document if a document matching the `query` is not found:

```
db.people.findAndModify( {
    query: { name: "Pascal", state: "active", rating: 25 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true,
    new: true
} )
```

The method returns the newly inserted document:

```
{
   "_id" : ObjectId("50f49ad6444c11ac2448a5d6"),
   "name" : "Pascal",
   "rating" : 25,
   "score" : 1,
   "state" : "active"
}
```

db.collection.findOne()

Definition

`db.collection.`**`findOne`**`(`*<criteria>*, *<projection>*`)`

Returns one document that satisfies the specified query criteria. 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 disk. In *capped collections*, natural order is the same as insertion order.

param document criteria Specifies query selection criteria using *query operators* (page 785).

param document projection Specifies the fields to return using *projection operators* (page 822). To return all fields in the matching document, omit this parameter.

Returns One document that satisfies the criteria specified as the first argument to this method. If you specify the `projection` argument, `findOne()` (page 955) returns a document that only contains the `projection` fields, and the `_id` field if you do not explicitly exclude the `_id` field.

Example The following is a prototype document for the projection argument:

```
{ field1: boolean, field2: boolean ... }
```

The `boolean` can take the following include or exclude values:

- `1` or `true` to include. The `findOne()` (page 955) 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 argument cannot contain both include and exclude specifications except for the exclusion of the `_id` field.

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()` (page 956) 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()` (page 956) items consist of the following fields:

`system.indexes.`**`v`**

Holds the version of the index.

The index version depends on the version of `mongod` (page 1049) that created the index. Before version 2.0 of MongoDB, the this value was 0; versions 2.0 and later use version 1.

`system.indexes.`**`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.

`system.indexes.`**`ns`**

The namespace context for the index.

`system.indexes.`**`name`**

A unique name for the index comprised of the field names and orders of all keys.

db.collection.getShardDistribution()

`db.collection.`**`getShardDistribution`**`()`

Returns

Prints the data distribution statistics for a *sharded* collection. You must call the `getShardDistribution()` (page 956) method on a sharded collection, as in the following example:

```
db.myShardedCollection.getShardDistribution()
```

In the following example, the collection has two shards. The output displays both the individual shard distribution information as well the total shard distribution:

```
Shard <shard-a> at <host-a>
 data : <size-a> docs : <count-a> chunks : <number of chunks-a>
 estimated data per chunk : <size-a>/<number of chunks-a>
 estimated docs per chunk : <count-a>/<number of chunks-a>

Shard <shard-b> at <host-b>
 data : <size-b> docs : <count-b> chunks : <number of chunks-b>
 estimated data per chunk : <size-b>/<number of chunks-b>
 estimated docs per chunk : <count-b>/<number of chunks-b>

Totals
 data : <stats.size> docs : <stats.count> chunks : <calc total chunks>
 Shard <shard-a> contains  <estDataPercent-a>% data, <estDocPercent-a>% docs in cluster, avg obj
 Shard <shard-b> contains  <estDataPercent-b>% data, <estDocPercent-b>% docs in cluster, avg obj
```

The output information displays:

- `<shard-x>` is a string that holds the shard name.
- `<host-x>` is a string that holds the host name(s).
- `<size-x>` is a number that includes the size of the data, including the unit of measure (e.g. `b`, `Mb`).
- `<count-x>` is a number that reports the number of documents in the shard.
- `<number of chunks-x>` is a number that reports the number of chunks in the shard.
- `<size-x>/<number of chunks-x>` is a calculated value that reflects the estimated data size per chunk for the shard, including the unit of measure (e.g. `b`, `Mb`).
- `<count-x>/<number of chunks-x>` is a calculated value that reflects the estimated number of documents per chunk for the shard.
- `<stats.size>` is a value that reports the total size of the data in the sharded collection, including the unit of measure.
- `<stats.count>` is a value that reports the total number of documents in the sharded collection.
- `<calc total chunks>` is a calculated number that reports the number of chunks from all shards, for example:

  ```
  <calc total chunks> = <number of chunks-a> + <number of chunks-b>
  ```

- `<estDataPercent-x>` is a calculated value that reflects, for each shard, the data size as the percentage of the collection's total data size, for example:

  ```
  <estDataPercent-x> = <size-x>/<stats.size>
  ```

- `<estDocPercent-x>` is a calculated value that reflects, for each shard, the number of documents as the percentage of the total number of documents for the collection, for example:

  ```
  <estDocPercent-x> = <count-x>/<stats.count>
  ```

- `stats.shards[ <shard-x> ].avgObjSize` is a number that reflects the average object size, including the unit of measure, for the shard.

  For example, the following is a sample output for the distribution of a sharded collection:

```
Shard shard-a at shard-a/MyMachine.local:30000,MyMachine.local:30001,MyMachine.local:30002
data : 38.14Mb docs : 1000003 chunks : 2
estimated data per chunk : 19.07Mb
estimated docs per chunk : 500001
```

```
Shard shard-b at shard-b/MyMachine.local:30100,MyMachine.local:30101,MyMachine.local:30102
data : 38.14Mb docs : 999999 chunks : 3
estimated data per chunk : 12.71Mb
estimated docs per chunk : 333333

Totals
data : 76.29Mb docs : 2000002 chunks : 5
Shard shard-a contains 50% data, 50% docs in cluster, avg obj size on shard : 40b
Shard shard-b contains 49.99% data, 49.99% docs in cluster, avg obj size on shard : 40b
```

See also:

Sharding (page 485)

db.collection.getShardVersion()

db.collection.**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.

db.collection.group()

Definition

db.collection.**group**(*{ key, reduce, initial, [keyf,] [cond,] finalize }*)

Groups documents in a collection by the specified keys and performs simple aggregation functions such as computing counts and sums. The method is analogous to a `SELECT .. GROUP BY` statement in SQL. The `group()` (page 958) method returns an array.

The `db.collection.group()` (page 958) accepts a single *document* that contains the following:

param document key The field or fields to group by.

param function reduce A function that group operation performs on the documents during the grouping operation. These functions may return a sum or a count. The function takes two arguments: the current document and the aggregate result for the previous documents in the group.

param document initial Initializes the aggregation result document.

param function keyf Alternative to the `key` field. Specifies a function that creates a "key object" for use as the grouping key. Use the `keyf` instead of `key` to group by calculated fields rather than existing document fields.

param document cond The selection criteria to determine which documents in the collection to process. If you omit the `cond` field, `db.collection.group()` (page 958) processes all the documents in the collection for the group operation.

param function finalize A function that runs each item in the result set before `db.collection.group()` (page 958) returns the final value. This function can either modify the result document or replace the entire result document.

The `db.collection.group()` (page 958) method is a shell wrapper for the `group` (page 836) command. However, the `db.collection.group()` (page 958) method takes the `keyf` field and the `reduce` field whereas the `group` (page 836) command takes the `$keyf` field and the `$reduce` field.

Warning:

- The `db.collection.group()` (page 958) method does not work with *sharded clusters*. Use the *aggregation framework* or *map-reduce* in *sharded environments*.
- The result set must fit within the *maximum BSON document size* (page 1139).
- In version 2.2, the returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. For group by operations that results in more than 20,000 unique groupings, use `mapReduce` (page 840). Previous versions had a limit of 10,000 elements.
- Prior to 2.4, the `db.collection.group()` (page 958) method took the `mongod` (page 1049) instance's JavaScript lock, which blocked all other JavaScript execution.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 840), `group` (page 836) commands, or `$where` (page 797) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

Examples The following examples assume an `orders` collection with documents of the following prototype:

```
{
  _id: ObjectId("5085a95c8fada716c89d0021"),
  ord_dt: ISODate("2012-07-01T04:00:00Z"),
  ship_dt: ISODate("2012-07-02T04:00:00Z"),
  item: { sku: "abc123",
          price: 1.99,
          uom: "pcs",
```

```
          qty: 25 }
}
```

Group by Two Fields The following example groups by the `ord_dt` and `item.sku` fields those documents that have `ord_dt` greater than `01/01/2011`:

```
db.orders.group( {
                   key: { ord_dt: 1, 'item.sku': 1 },
                   cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                   reduce: function ( curr, result ) { },
                   initial: { }
                } )
```

The result is an array of documents that contain the group by fields:

```
[ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123"},
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456"},
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123"},
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456"},
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123"},
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456"} ]
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate the Sum The following example groups by the `ord_dt` and `item.sku` fields, those documents that have `ord_dt` greater than `01/01/2011` and calculates the sum of the `qty` field for each grouping:

```
db.orders.group( {
                   key: { ord_dt: 1, 'item.sku': 1 },
                   cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                   reduce: function ( curr, result ) {
                               result.total += curr.item.qty;
                           },
                   initial: { total : 0 }
                } )
```

The result is an array of documents that contain the group by fields and the calculated aggregation field:

```
[ { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "bcd123", "total" : 10 },
  { "ord_dt" : ISODate("2012-07-01T04:00:00Z"), "item.sku" : "efg456", "total" : 10 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "efg456", "total" : 15 },
  { "ord_dt" : ISODate("2012-06-01T04:00:00Z"), "item.sku" : "ijk123", "total" : 20 },
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc123", "total" : 45 },
  { "ord_dt" : ISODate("2012-05-01T04:00:00Z"), "item.sku" : "abc456", "total" : 25 },
```

```
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc123", "total" : 25 },
  { "ord_dt" : ISODate("2012-06-08T04:00:00Z"), "item.sku" : "abc456", "total" : 25 } ]
```

The method call is analogous to the SQL statement:

```
SELECT ord_dt, item_sku, SUM(item_qty) as total
FROM orders
WHERE ord_dt > '01/01/2012'
GROUP BY ord_dt, item_sku
```

Calculate Sum, Count, and Average The following example groups by the calculated `day_of_week` field, those documents that have `ord_dt` greater than `01/01/2011` and calculates the sum, count, and average of the `qty` field for each grouping:

```
db.orders.group( {
                   keyf: function(doc) {
                             return { day_of_week: doc.ord_dt.getDay() } ; },
                   cond: { ord_dt: { $gt: new Date( '01/01/2012' ) } },
                   reduce: function ( curr, result ) {
                              result.total += curr.item.qty;
                              result.count++;
                           },
                   initial: { total : 0, count: 0 },
                   finalize: function(result) {
                               var weekdays = [ "Sunday", "Monday", "Tuesday",
                                                "Wednesday", "Thursday",
                                                "Friday", "Saturday" ];

                               result.day_of_week = weekdays[result.day_of_week];
                               result.avg = Math.round(result.total / result.count);

                   }
                } )
```

The result is an array of documents that contain the group by fields and the calculated aggregation field:

```
[ { "day_of_week" : "Sunday", "total" : 70, "count" : 4, "avg" : 18 },
  { "day_of_week" : "Friday", "total" : 110, "count" : 6, "avg" : 18 },
  { "day_of_week" : "Tuesday", "total" : 70, "count" : 3, "avg" : 23 } ]
```

See also:

Aggregation Framework (page 247)

db.collection.insert()

Definition

`db.collection.insert(`*document|array*`)`

Inserts a document or an array of documents into a collection.

Changed in version 2.2: The `insert()` (page 961) method can accept an array of documents to perform a bulk insert of the documents into the collection.

param document,array document A document or array of documents to insert into the collection.

The `insert()` (page 961) method has the following behaviors:

- If the collection does not exist, then the `insert()` (page 961) 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 (page 1049) will create and populate the _id if the driver or application does not.
- If the document specifies a new field, then the insert() (page 961) method inserts the document with the new field. This requires no changes to the data model for the collection or the existing documents.

Examples The following are examples of the insert() (page 961) 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() (page 961) 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 (page 1049) 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() (page 961) 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 (page 1049) will add the _id field and generate the ObjectId.

- To insert multiple documents, pass an array of documents to the insert() (page 961) 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.isCapped()

`db.collection.isCapped()`

Returns Returns `true` if the collection is a *capped collection*, otherwise returns `false`.

See also:

Capped Collections (page 578)

db.collection.mapReduce()

`db.collection.mapReduce(`*map*, *reduce*, *{<out>*, *<query>*, *<sort>*, *<limit>*, *<finalize>*, *<scope>*, *<jsMode>*, *<verbose>}*`)`

The `db.collection.mapReduce()` (page 963) method provides a wrapper around the `mapReduce` (page 840) command.

```
db.collection.mapReduce(
                         <map>,
                         <reduce>,
                         {
                           out: <collection>,
                           query: <document>,
                           sort: <document>,
                           limit: <number>,
                           finalize: <function>,
                           scope: <document>,
                           jsMode: <boolean>,
                           verbose: <boolean>
                         }
                       )
```

`db.collection.mapReduce()` (page 963) takes the following parameters:

Parameters

- **map** – A JavaScript function that associates or "maps" a `value` with a `key` and emits the `key` and value `pair`.

  The `map` function processes every input document for the map-reduce operation. However, the `map` function can call emit any number of times, including 0, for each input document. The map-reduce operation groups the emitted `value` objects by the `key` and passes these groupings to the `reduce` function. See below for requirements for the `map` function.

- **reduce** – A JavaScript function that "reduces" to a single object all the `values` associated with a particular `key`.

  The `reduce` function accepts two arguments: `key` and `values`. The `values` argument is an array whose elements are the `value` objects that are "mapped" to the `key`. See below for requirements for the `reduce` function.

- **out** – New in version 1.8.

  Specifies the location of the result of the map-reduce operation. You can output to a collection, output to a collection with an action, or output inline. You may output to a collection when performing map reduce operations on the primary members of the set; on *secondary* members you may only use the `inline` output.

- **query** – Optional. Specifies the selection criteria using *query operators* (page 785) for determining the documents input to the `map` function.

- **sort** – Optional. Sorts the *input* documents. This option is useful for optimization. For example, specify the sort key to be the same as the emit key so that there are fewer reduce operations. The sort key must be in an existing index for this collection.
- **limit** – Optional. Specifies a maximum number of documents to return from the collection.
- **finalize** – Optional. A JavaScript function that follows the `reduce` method and modifies the output.

  The `finalize` function receives two arguments: `key` and `reducedValue`. The `reducedValue` is the value returned from the `reduce` function for the `key`.
- **scope** (*document*) – Optional. Specifies global variables that are accessible in the `map` , `reduce` and the `finalize` functions.
- **jsMode** (*Boolean*) – New in version 2.0.

  Optional. Specifies whether to convert intermediate data into BSON format between the execution of the `map` and `reduce` functions.

  If `false`:

  – Internally, MongoDB converts the JavaScript objects emitted by the `map` function to BSON objects. These BSON objects are then converted back to JavaScript objects when calling the `reduce` function.

  – The map-reduce operation places the intermediate BSON objects in temporary, on-disk storage. This allows the map-reduce operation to execute over arbitrarily large data sets.

  If `true`:

  – Internally, the JavaScript objects emitted during `map` function remain as JavaScript objects. There is no need to convert the objects for the `reduce` function, which can result in faster execution.

  – You can only use `jsMode` for result sets with fewer than 500,000 distinct `key` arguments to the mapper's `emit()` function.

  The `jsMode` defaults to false.
- **verbose** (*Boolean*) – Optional. Specifies whether to include the `timing` information in the result information. The `verbose` defaults to `true` to include the `timing` information.

Note: Changed in version 2.4.

In MongoDB 2.4, `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 840), `group` (page 836) commands, or `$where` (page 797) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

Requirements for the `map` Function The `map` function has the following prototype:

```
function() {
   ...
   emit(key, value);
}
```

The `map` function exhibits the following behaviors:

- In the `map` function, reference the current document as `this` within the function.
- The `map` function should *not* access the database for any reason.
- The `map` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `emit(key,value)` function associates the `key` with a `value`.
  - A single emit can only hold half of MongoDB's *maximum BSON document size* (page 1139).
  - The `map` function can call `emit(key,value)` any number of times, including 0, per each input document.

    The following `map` function may call `emit(key,value)` either 0 or 1 times depending on the value of the input document's `status` field:

    ```
    function() {
        if (this.status == 'A')
            emit(this.cust_id, 1);
    }
    ```

    The following `map` function may call `emit(key,value)` multiple times depending on the number of elements in the input document's `items` field:

    ```
    function() {
        this.items.forEach(function(item){ emit(item.sku, 1); });
    }
    ```

- The `map` function can access the variables defined in the `scope` parameter.

Requirements for the `reduce` Function The `reduce` function has the following prototype:

```
function(key, values) {
   ...
   return result;
}
```

The `reduce` function exhibits the following behaviors:

- The `reduce` function should *not* access the database, even to perform read operations.
- The `reduce` function should *not* affect the outside system.
- MongoDB will **not** call the `reduce` function for a key that has only a single value.
- MongoDB can invoke the `reduce` function more than once for the same key. In this case, the previous output from the `reduce` function for that key will become one of the input values to the next `reduce` function invocation for that key.
- The `reduce` function can access the variables defined in the `scope` parameter.

Because it is possible to invoke the `reduce` function more than once for the same key, the following properties need to be true:

- the *type* of the return object must be **identical** to the type of the `value` emitted by the `map` function to ensure that the following operations is true:

  ```
  reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
  ```

- the `reduce` function must be *idempotent*. Ensure that the following statement is true:

  ```
  reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
  ```

- the order of the elements in the `valuesArray` should not affect the output of the `reduce` function, so that the following statement is true:

  ```
  reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
  ```

`out` Options You can specify the following options for the `out` parameter:

Output to a Collection

```
out: <collectionName>
```

Output to a Collection with an Action This option is only available when passing `out` a collection that already exists. This option is not available on secondary members of replica sets.

```
out: { <action>: <collectionName>
        [, db: <dbName>]
        [, sharded: <boolean> ]
        [, nonAtomic: <boolean> ] }
```

When you output to a collection with an action, the `out` has the following parameters:

- `<action>`: Specify one of the following actions:

  - `replace`

    Replace the contents of the `<collectionName>` if the collection with the `<collectionName>` exists.

  - `merge`

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, *overwrite* that existing document.

  - `reduce`

    Merge the new result with the existing result if the output collection already exists. If an existing document has the same key as the new result, apply the `reduce` function to both the new and the existing documents and overwrite the existing document with the result.

- `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* you have enabled sharding on output database, the map-reduce operation will shard the output collection using the `_id` field as the shard key.

- `nonAtomic`:

  New in version 2.2.

  Optional. Specify output operation as non-atomic and is valid *only* for `merge` and `reduce` output modes which may take minutes to execute.

  If `nonAtomic` is `true`, the post-processing step will prevent MongoDB from locking the database; however, other clients will be able to read intermediate states of the output collection. Otherwise the map reduce operation must lock the database during post-processing.

Output Inline Perform the map-reduce operation in memory and return the result. This option is the only available option for `out` on secondary members of replica sets.

```
out: { inline: 1 }
```

The result must fit within the *maximum size of a BSON document* (page 1139).

Requirements for the `finalize` Function The `finalize` function has the following prototype:

```
function(key, reducedValue) {
   ...
   return modifiedObject;
}
```

The `finalize` function receives as its arguments a `key` value and the `reducedValue` from the `reduce` function. Be aware that:

- The `finalize` function should *not* access the database for any reason.
- The `finalize` function should be pure, or have *no* impact outside of the function (i.e. side effects.)
- The `finalize` function can access the variables defined in the `scope` parameter.

Map-Reduce Examples Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer Perform map-reduce operation on the `orders` collection to group by the `cust_id`, and for each `cust_id`, calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

   ```
   var mapFunction1 = function() {
                          emit(this.cust_id, this.price);
                      };
   ```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:
   - The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
   - The function reduces the `valuesPrice` array to the sum of its elements.

   ```
   var reduceFunction1 = function(keyCustId, valuesPrices) {
                             return Array.sum(valuesPrices);
                         };
   ```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

   ```
   db.orders.mapReduce(
                        mapFunction1,
                        reduceFunction1,
                        { out: "map_reduce_example" }
                      )
   ```

   This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item In this example you will perform a map-reduce operation on the `orders` collection, for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and for each `sku` calculates the number of orders and the total quantity ordered. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:

- In the function, `this` refers to the document that the map-reduce operation is processing.
- For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
                                         qty: this.items[idx].qty
                                       };
                           emit(key, value);
                       }
                    };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:
   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
                     reducedVal = { count: 0, qty: 0 };

                     for (var idx = 0; idx < countObjVals.length; idx++) {
                         reducedVal.count += countObjVals[idx].count;
                         reducedVal.qty += countObjVals[idx].qty;
                     }

                     return reducedVal;
                  };
```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

                       reducedVal.avg = reducedVal.qty/reducedVal.count;

                       return reducedVal;

                    };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                       out: { merge: "map_reduce_example" },
                       query: { ord_date:
                                  { $gt: new Date('01/01/2012') }
                              },
                       finalize: finalizeFunction2
```

```
                        }
                      )
```

This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

For more information and examples, see the *Map-Reduce* (page 291) page and *Perform Incremental Map-Reduce* (page 293).

See also:

- *Troubleshoot the Map Function* (page 297)
- *Troubleshoot the Reduce Function* (page 298)
- `mapReduce` (page 840) command
- *Aggregation Framework* (page 247)

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()

Definition

`db.collection.`**`remove`**`(`*query*, *justOne*`)`

Removes documents from a collection.

param document query Specifies deletion criteria using *query operators* (page 785). To delete all documents in a collection, omit this parameter or pass an empty document (`{}`).

param boolean justOne To limit the deletion to just one document, set to `true`. The default value is `false`.

Note: You cannot use the `remove()` (page 970) method with a *capped collection*.

Examples The following are examples of the `remove` (page 970) method.

- To remove all documents in a collection, call the `remove` (page 970) 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` (page 970) 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` (page 970) method with the `query` criteria and the `justOne` parameter set to `true` or `1`:

  ```
  db.products.remove( { qty: { $gt: 20 } }, true )
  ```

  This operation removes the first document from the collection `products` where `qty` is greater than `20`.

Note: If the `query` argument to the `remove()` (page 970) 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 `$isolated` (page 822) isolation operator, effectively isolating the delete operation and blocking other write operations during the delete. To isolate the query, include `$isolated: 1` in the `query` parameter as in the following example:

```
db.products.remove( { qty: { $gt: 20 }, $isolated: 1 } )
```

db.collection.renameCollection()

Definition

`db.collection.renameCollection(target, string)`

Renames a collection. Provides a wrapper for the `renameCollection` (page 882) *database command*.

param string target The new name of the collection. Enclose the string in quotes.

param boolean dropTarget If `true`, `mongod` (page 1049) drops the *target* of `renameCollection` (page 882) prior to renaming the collection.

Example Call the `db.collection.renameCollection()` (page 971) method on a collection object. 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.

Limitations The method has the following limitations:

- `db.collection.renameCollection()` (page 971) cannot move a collection between databases. Use `renameCollection` (page 882) for these rename operations.
- `db.collection.renameCollection()` (page 971) cannot operation on sharded collections.

The `db.collection.renameCollection()` (page 971) method operates within a collection by changing the metadata associated with a given collection.

Refer to the documentation `renameCollection` (page 882) for additional warnings and messages.

> **Warning:** The `db.collection.renameCollection()` (page 971) method and `renameCollection` (page 882) command will invalidate open cursors which interrupts queries that are currently returning data.

db.collection.save()

`db.collection.save(document)`

Updates an existing document or inserts a new document, depending on the specified document.

The `save()` (page 972) method takes the following parameter:

param document document A document to save to the collection.

If the document does not contain an *_id* field, then the `save()` (page 972) 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()` (page 972) method performs an `upsert` querying the collection on the `_id` field:

- If a document does not exist with the specified `_id` value, the `save()` (page 972) method performs an insert with the specified fields in the document.
- If a document exists with the specified `_id` value, the `save()` (page 972) method performs an update, replacing all field in the existing record with the fields from the document.

Examples The following are examples of the `save()` (page 972) method.

Example

Pass to the `save()` (page 972) 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` (page 1049) will add the `_id` field and generate the `ObjectId`.

Example

Pass to the `save()` (page 972) 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` (page 1049) will add the `_id` field and generate the `ObjectId`.

Example

Pass to the `save()` (page 972) 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()

Definition

`db.collection.stats(scale)`

Returns statistics about the collection. The method includes the following parameter:

param number scale The scale used in the output to display the sizes of items. By default, output displays sizes in bytes. To display kilobytes rather than bytes, specify a `scale` value of `1024`.

Returns A *document* containing statistics that reflecting the state of the specified collection.

The `stats()` (page 973) method provides a wrapper around the database command `collStats` (page 900).

Example The following operation returns stats on the `people` collection:

```
db.people.stats()
```

See also:

collStats (page 900) for an overview of the output of this command.

db.collection.storageSize()

`db.collection.storageSize()`

Returns The total amount of storage allocated to this collection for document storage. Provides a wrapper around the `storageSize` (page 901) field of the `collStats` (page 900) (i.e. `db.collection.stats()` (page 973)) output.

db.collection.totalSize()

`db.collection.totalSize()`

Returns The total size of the data in the collection plus the size of every indexes on the collection.

db.collection.totalIndexSize()

`db.collection.totalIndexSize()`

Returns The total size of all indexes for the collection. This method provides a wrapper around the `totalIndexSize` (page 902) output of the `collStats` (page 900) (i.e. `db.collection.stats()` (page 973)) operation.

db.collection.update()

Definition

db.collection.**update**(*query*, *update*[, *options*])

Modifies an existing document or documents in a collection. `update()` (page 974) takes the following parameters:

param document query The selection criteria for the update. Use the same *query selectors* (page 785) as used in the `find()` (page 951) method.

param document update The modifications to apply. For details see *Update Parameter* (page 974) after this table.

param document options New in version 2.2: Specifies whether to perform an *upsert* and/or a multiple update. Use the `options` parameter instead of the individual `upsert` and `multi` parameters.

param Boolean upsert Specifies whether to create a new document if no document matches the query criteria. When `upsert` is set to `true`, the `update()` (page 974) method either updates an existing document, or, if no document matches the criteria, inserts a new document. When `upsert` is set to `false`, `update()` (page 974) does not insert a new document if no match is found. The default value is `false`. For additional information, see *Upsert Parameter* (page 975).

param Boolean multi Specifies whether to update multiple documents that meet the `query` criteria. When `multi` is set to `true`, the `update()` (page 974) method updates all documents that meet the criteria. When `multi` is set to `false`, the method updates one document. The default value is `false`. For additional information, see *Multi Parameter* (page 975).

`update()` (page 974) updates a single document by default. 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` (page 1066) 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` (page 1066) shell, `upsert` and `multi` were positional boolean options:

```
db.collection.update(query, update, <upsert>, <multi>)
```

Although the update operation may apply mostly to updating the values of the fields, `update()` (page 974) can also modify the name of the `field` in a document using the `$rename` (page 810) operator.

Update Parameter If the `update` parameter contains any *update operator* (page 810) expressions, such as the `$set` (page 814) operator expression, then:

- the `update` parameter must contain only update operator expressions.

New in version 2.2.

- `update()` (page 974) updates only the corresponding fields in the document.

If the `update` parameter consists only of `field: value` expressions, then:

- `update()` (page 974) *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* .
- `update()` (page 974) updates cannot update multiple documents.

Upsert Parameter If upsert is set to true and no document matches the criteria, update() (page 974) inserts a new document with the fields and values of the update parameter. If the update included only update operators, update() (page 974) inserts the query parameter as well .

In version 2.2 of the mongo (page 1066) shell, you may also specify upsert in the options parameter.

With upsert, update() (page 974) inserts a *single* document.

Multi Parameter When multi is set to true, the update() (page 974) method updates all documents that meet the query criteria. In version 2.2 of the mongo (page 1066) shell, you may also specify multi in the options parameter.

The multi update operation may interleave with other write operations. For unsharded collections, you can override this behavior with the $isolated (page 822) isolation operator, which isolates the update operation and blocks other write operations during the update.

Examples The following examples use the 2.2 interface to specify options in the document form.

Update Specific Fields in a Document Call update() (page 974) with an update parameter using field: value pairs and expressions using *update operators* (page 810) 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.

Replace All Fields in a Document To replace all the fields in a document with the document as specified in the update parameter, call the update() (page 974) 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.

Update Multiple Documents Call the update() (page 974) method and specify the multi option, as in the following:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, { multi: true } )
```

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() (page 974) method with the multi parameter:

```
db.products.update( { item: "book", qty: { $gt: 5 } }, { $set: { x: 6, y: 15 } }, false, true )
```

Upsert To update a document or to insert a new document if no document matches the query criteria, call the update() (page 974) and specify the upsert option, as in the following:

```
db.products.update( { item: "magazine", qty: { $gt: 5 } }, { $set: { x: 25, y: 50 } }, { upsert: tru
```

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()

Description

`db.collection.`**`validate`**(*full*)

Validates a collection. The method scans a collection's data structures for correctness and returns a single *document* that describes the relationship between the logical collection and the physical representation of the data.

The `validate()` (page 976) method has the following parameter:

param Boolean full Specify `true` to enable a full validation and to return full statistics. MongoDB disables full validation by default because it is a potentially resource-intensive operation.

The `validate()` (page 976) method output provides an 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.

The `validate()` (page 976) method is a wrapper around the `validate` (page 908) *database command*.

See also:

validate (page 908)

55.3.2 Cursor

Cursor Methods

Name	Description
`cursor.addOption()` (page 977)	Adds special wire protocol flags that modify the behavior of the query.'
`cursor.batchSize()` (page 978)	Controls the number of documents MongoDB will return to the client in a single network message.
`cursor.count()` (page 978)	Returns a count of the documents in a cursor.
`cursor.explain()` (page 979)	Reports on the query execution plan, including index use, for a cursor.
`cursor.forEach()` (page 984)	Applies a JavaScript function for every document in a cursor.
`cursor.hasNext()` (page 985)	Returns true if the cursor has documents and can be iterated.
`cursor.hint()` (page 985)	Forces MongoDB to use a specific index for a query.
`cursor.limit()` (page 985)	Constrains the size of a cursor's result set.
`cursor.map()` (page 986)	Applies a function to each document in a cursor and collects the return values in an array.
`cursor.max()` (page 986)	Specifies an exclusive upper index bound for a cursor. For use with `cursor.hint()` (page 985)
`cursor.min()` (page 987)	Specifies an inclusive lower index bound for a cursor. For use with `cursor.hint()` (page 985)
`cursor.next()` (page 989)	Returns the next document in a cursor.
`cursor.objsLeftInBatch()` (page 989)	Returns the number of documents left in the current cursor batch.
`cursor.readPref()` (page 989)	Specifies a *read preference* to a cursor to control how the client directs queries to a *replica set*.
`cursor.showDiskLoc()` (page 990)	Returns a cursor with modified documents that include the on-disk location of the document.
`cursor.size()` (page 990)	Returns a count of the documents in the cursor after applying `skip()` (page 990) and `limit()` (page 985) methods.
`cursor.skip()` (page 990)	Returns a cursor that begins returning results only after passing or skipping a number of documents.
`cursor.snapshot()` (page 990)	Forces the cursor to use the index on the `_id` field. Ensures that the cursor returns each document, with regards to the value of the `_id` field, only once.
`cursor.sort()` (page 991)	Returns results ordered according to a sort specification.
`cursor.toArray()` (page 992)	Returns an array that contains all documents returned by the cursor.

cursor.addOption()

Definition

`cursor.addOption(flag)`

Adds `OP_QUERY` wire protocol flags, such as the `tailable` flag, to change the behavior of queries.

The `cursor.addOption()` (page 977) method has the following parameter:

param flag flag `OP_QUERY` wire protocol flag. See MongoDB wire protocol for more information on MongoDB Wire Protocols and the OP_QUERY flags. For the `mongo` (page 1066) shell, you can use *cursor flags* (page 51). For the driver-specific list, see your *driver documentation* (page 575).

Example The following example adds the `DBQuery.Option.tailable` flag and the `DBQuery.Option.awaitData` flag to ensure that the query returns a tailable cursor. The sequence creates a cursor that will wait for few seconds after returning the full result set so that it can capture and return additional data added during the query:

```
var t = db.myCappedCollection;
var cursor = t.find().addOption(DBQuery.Option.tailable).
                      addOption(DBQuery.Option.awaitData)
```

Warning: Adding incorrect wire protocol flags can cause problems and/or extra server load.

cursor.batchSize()

Definition

`cursor.batchSize(size)`

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, as the `mongo` (page 1066) shell and most *drivers* (page 575) return results as if MongoDB returned a single batch.

The `batchSize()` (page 978) method takes the following parameter:

param integer 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()` (page 985) method.

Example The following example sets the batch size for the results of a query (i.e. `find()` (page 951)) to `10`. The `batchSize()` (page 978) method does not change the output in the `mongo` (page 1066) shell, which, by default, iterates over the first 20 documents.

```
db.inventory.find().batchSize(10)
```

cursor.count()

Definition

`cursor.count()`

Counts the number of documents referenced by a cursor. Append the `count()` (page 978) method to a `find()` (page 951) query to return the number of matching documents. The operation does not perform the query but instead counts the results that would be returned by the query.

The `count()` (page 978) method has the following prototype form:

```
db.collection.find().count()
```

The `count()` (page 978) method has the following parameter:

param Boolean applySkipLimit Specifies whether to consider the effects of the `cursor.skip()` (page 990) and `cursor.limit()` (page 985) methods in the count. By default, the `count()` (page 978) method ignores the effects of the `cursor.skip()` (page 990) and `cursor.limit()` (page 985). Set `applySkipLimit` to `true` to consider the effect of these methods.

See also:

`cursor.size()` (page 990)

MongoDB also provides the shell wrapper `db.collection.count()` (page 946) for the `db.collection.find().count()` construct.

Examples The following are examples of the `count()` (page 978) method.

Example

Count the number of all documents in the `orders` collection:

```
db.orders.find().count()
```

Example

Count the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')`:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).count()
```

Example

Count the number of the documents in the `orders` collection with the field `ord_dt` greater than `new Date('01/01/2012')` *taking into account* the effect of the `limit(5)`:

```
db.orders.find( { ord_dt: { $gt: new Date('01/01/2012') } } ).limit(5).count(true)
```

---

cursor.explain()

Definition

`cursor.explain()`

Provides information on the query plan. The query plan is the plan the server uses to find the matches for a query. This information may be useful when optimizing a query. The `explain()` (page 979) method returns a document that describes the process used to return the query results.

The `explain()` (page 979) method has the following form:

```
db.collection.find().explain()
```

The `explain()` (page 979) method has the following parameter:

param Boolean verbose Specifies the level of detail to include in the output. If `true` or `1`, includes the `allPlans` and `oldPlan` fields in the output.

For an explanation of output, see *Explain on Queries on Sharded Collections* (page 981) and *Core Explain Output Fields* (page 982).

The `explain()` (page 979) method runs the actual query to determine the result. Although there are some differences between running the query with `explain` (page 979) and running without, generally, the performance will be similar between the two. So, if the query is slow, the `explain` (page 979) operation is also slow.

Additionally, the `explain` (page 979) operation reevaluates a set of candidate query plans, which may cause the `explain` (page 979) operation to perform differently than a normal query. As a result, these operations generally provide an accurate account of *how* MongoDB would perform the query, but do not reflect the length of these queries.

To determine the performance of a particular index, you can use `hint()` (page 985) and in conjunction with `explain()` (page 979), as in the following example:

```
db.products.find().hint( { type: 1 } ).explain()
```

When you run `explain` (page 979) with `hint()` (page 985), the query optimizer does not reevaluate the query plans.

Note: In some situations, the `explain()` (page 979) operation may differ from the actual query plan used by MongoDB in a normal query. The `explain()` (page 979) operation evaluates the set of query plans and reports on the winning plan for the query. In normal operations the query optimizer caches winning query plans and uses them for similar related queries in the future. As a result MongoDB may sometimes select query plans from the cache that are different from the plan displayed using `explain` (page 979).

See also:

- `$explain` (page 828)
- *Optimization Strategies for MongoDB* (page 576) page for information regarding optimization strategies.
- *Analyze Performance of Database Operations* (page 175) tutorial for information regarding the database profile.
- *Current Operation Reporting* (page 998)

Explain Results

Explain on Queries on Unsharded Collections For queries on unsharded collections, `explain()` (page 979) returns the following core information.

```
{
  "cursor" : "<Cursor Type and Index>",
  "isMultiKey" : <boolean>,
  "n" : <num>,
  "nscannedObjects" : <num>,
  "nscanned" : <num>,
  "nscannedObjectsAllPlans" : <num>,
  "nscannedAllPlans" : <num>,
  "scanAndOrder" : <boolean>,
  "indexOnly" : <boolean>,
  "nYields" : <num>,
  "nChunkSkips" : <num>,
  "millis" : <num>,
  "indexBounds" : { <index bounds> },
  "allPlans" : [
                 { "cursor" : "<Cursor Type and Index>",
                   "n" : <num>,
                   "nscannedObjects" : <num>,
```

```
                   "nscanned" : <num>,
                   "indexBounds" : { <index bounds> }
                 },
                 ...
              ],
  "oldPlan" : {
                "cursor" : "<Cursor Type and Index>",
                "indexBounds" : { <index bounds> }
              }
  "server" : "<host:port>",
}
```

For details on the fields, see *Core Explain Output Fields* (page 982).

Explain on `$or` Queries Queries with `$or` (page 790) operator execute each clause of the `$or` (page 790) expression in parallel and can use separate indexes on the individual clauses. If the query uses indexes on any or all of the query's clause, `explain()` (page 979) contains *output* (page 982) for each clause as well as the cumulative data for the entire query:

```
{
   "clauses" : [
                  {
                     <core explain output>
                  },
                  {
                     <core explain output>
                  },
                  ...
               ],
   "n" : <num>,
   "nscannedObjects" : <num>,
   "nscanned" : <num>,
   "nscannedObjectsAllPlans" : <num>,
   "nscannedAllPlans" : <num>,
   "millis" : <num>,
   "server" : "<host:port>"
}
```

For details on the fields, see *$or Query Output Fields* (page 984) and *Core Explain Output Fields* (page 982).

Explain on Queries on Sharded Collections For queries on sharded collections, `explain()` (page 979) returns information for each shard the query accesses. For queries on unsharded collections, see *Core Explain Output Fields* (page 982).

For queries on a sharded collection, the output contains the *Core Explain Output Fields* (page 982) for each accessed shard and *cumulative shard information* (page 984):

```
{
   "clusteredType" : "<Shard Access Type>",
   "shards" : {

                "<shard1>" : [
                               {
                                 <core explain output>
                               }
                             ],
                "<shard2>" : [
```

```
                                {
                                 <core explain output>
                                }
                              ],
                 ...
               },
   "millisShardTotal" : <num>,
   "millisShardAvg" : <num>,
   "numQueries" : <num>,
   "numShards" : <num>,
   "cursor" : "<Cursor Type and Index>",
   "n" : <num>,
   "nChunkSkips" : <num>,
   "nYields" : <num>,
   "nscanned" : <num>,
   "nscannedAllPlans" : <num>,
   "nscannedObjects" : <num>,
   "nscannedObjectsAllPlans" : <num>,
   "millis" : <num>
}
```

For details on these fields, see *Core Explain Output Fields* (page 982) for each accessed shard and *Sharded Collections Output Fields* (page 984).

Explain Output Fields

Core Explain Output Fields This section explains output for queries on collections that are *not sharded*. For queries on sharded collections, see *Explain on Queries on Sharded Collections* (page 981).

explain.**cursor**

`cursor` (page 982) is a string that reports the type of cursor used by the query operation:

- `BasicCursor` indicates a full collection scan.
- `BtreeCursor` indicates that the query used an index. The cursor includes name of the index. When a query uses an index, the output of `explain()` (page 979) includes `indexBounds` (page 983) details.
- `GeoSearchCursor` indicates that the query used a geospatial index.

explain.**isMultiKey**

`isMultiKey` (page 982) is a boolean. When `true`, the query uses a *multikey index* (page 312), where one of the fields in the index holds an array.

explain.**n**

`n` (page 982) is a number that reflects the number of documents that match the query selection criteria.

explain.**nscannedObjects**

Specifies the total number of documents scanned during the query. The `nscannedObjects` (page 982) may be lower than `nscanned` (page 982), such as if the index *covers* (page 322) a query. See `indexOnly` (page 983). Additionally, the `nscannedObjects` (page 982) may be lower than `nscanned` (page 982) in the case of multikey index on an array field with duplicate documents.

explain.**nscanned**

Specifies the total number of documents or index entries scanned during the database operation. You want `n` (page 982) and `nscanned` (page 982) to be close in value as possible. The `nscanned` (page 982) value may be higher than the `nscannedObjects` (page 982) value, such as if the index *covers* (page 322) a query. See `indexOnly` (page 983).

explain.**nscannedObjectsAllPlans**

New in version 2.2.

nscannedObjectsAllPlans (page 982) is a number that reflects the total number of documents scanned for all query plans during the database operation.

explain.**nscannedAllPlans**

New in version 2.2.

nscannedAllPlans (page 983) is a number that reflects the total number of documents or index entries scanned for all query plans during the database operation.

explain.**scanAndOrder**

scanAndOrder (page 983) is a boolean that is `true` when the query **cannot** use the order of documents in the index for returning sorted results.

When `false`, MongoDB must sort the documents after it receives the documents from a cursor.

explain.**indexOnly**

indexOnly (page 983) is a boolean value that returns `true` when the query is *covered* (page 322) by the index indicated in the cursor (page 982) field. When an index covers a query, MongoDB can both match the *query conditions* (page 42) **and** return the results using only the index because:

- all the fields in the *query* (page 42) are part of that index, **and**
- all the fields returned in the results set are in the same index.

explain.**nYields**

nYields (page 983) is a number that reflects the number of times this query yielded the read lock to allow waiting writes execute.

explain.**nChunkSkips**

nChunkSkips (page 983) is a number that reflects the number of documents skipped because of active chunk migrations in a sharded system. Typically this will be zero. A number greater than zero is ok, but indicates a little bit of inefficiency.

explain.**millis**

millis (page 983) is a number that reflects the time in milliseconds to complete the query.

explain.**indexBounds**

indexBounds (page 983) is a document that contains the lower and upper index key bounds. This field resembles one of the following:

```
"indexBounds" : {
                    "start" : { <index key1> : <value>, ...  },
                    "end" : { <index key1> : <value>, ... }
                },

"indexBounds" : { "<field>" : [ [ <lower bound>, <upper bound> ] ],
                  ...
                }
```

explain.**allPlans**

allPlans (page 983) is an array that holds the list of plans the query optimizer runs in order to select the index for the query. Displays only when the `<verbose>` parameter to `explain()` (page 979) is `true` or `1`.

explain.**oldPlan**

New in version 2.2.

oldPlan (page 983) is a document value that contains the previous plan selected by the query optimizer for the query. Displays only when the `<verbose>` parameter to `explain()` (page 979) is `true` or `1`.

explain.**server**

New in version 2.2.

`server` (page 983) is a string that reports the MongoDB server.

`$or` Query Output Fields

explain.**clauses**

`clauses` (page 984) is an array that holds the *Core Explain Output Fields* (page 982) information for each clause of the `$or` (page 790) expression. `clauses` (page 984) is only included when the clauses in the `$or` (page 790) expression use indexes.

Sharded Collections Output Fields

explain.**clusteredType**

`clusteredType` (page 984) is a string that reports the access pattern for shards. The value is:

- `ParallelSort`, if the `mongos` (page 1061) queries shards in parallel.
- `SerialServer`, if the `mongos` (page 1061) queries shards sequentially.

explain.**shards**

`shards` (page 984) contains fields for each shard in the cluster accessed during the query. Each field holds the *Core Explain Output Fields* (page 982) for that shard.

explain.**millisShardTotal**

`millisShardTotal` (page 984) is a number that reports the total time in milliseconds for the query to run on the shards.

explain.**millisShardAvg**

`millisShardAvg` (page 984) is a number that reports the average time in millisecond for the query to run on each shard.

explain.**numQueries**

`numQueries` (page 984) is a number that reports the total number of queries executed.

explain.**numShards**

`numShards` (page 984) is a number that reports the total number of shards queried.

cursor.forEach()

Description

cursor.**forEach**(*function*)

Iterates the cursor to apply a JavaScript `function` to each document from the cursor.

The `forEach()` (page 984) method has the following prototype form:

```
db.collection.find().forEach(<function>)
```

The `forEach()` (page 984) method has the following parameter:

param JavaScript function A JavaScript function to apply to each document from the cursor. The `<function>` signature includes a single argument that is passed the current document to process.

Example The following example invokes the `forEach()` (page 984) method on the cursor returned by `find()` (page 951) to print the name of each user in the collection:

```
db.users.find().forEach( function(myDoc) { print( "user: " + myDoc.name ); } );
```

See also:

`cursor.map()` (page 986) for similar functionality.

cursor.hasNext()

`cursor.hasNext()`

Returns Boolean.

`cursor.hasNext()` (page 985) returns `true` if the cursor returned by the `db.collection.find()` (page 951) query can iterate further to return more documents.

cursor.hint()

Definition

`cursor.hint(index)`

Call this method on a query to override MongoDB's default index selection and query optimization process. Use `db.collection.getIndexes()` (page 956) to return the list of current indexes on a collection.

The `cursor.hint()` (page 985) method has the following parameter:

param string,document index The index to "hint" or force MongoDB to use when performing the query. Specify the index either by the index name or by the index specification document.

See

Index Specification Documents (page 66) for information on index specification documents.

Example The following example returns all documents in the collection named `users` using the index on the `age` field.

```
db.users.find().hint( { age: 1 } )
```

You can also specify the index using the index name:

```
db.users.find().hint( "age_1" )
```

See also:

`$hint` (page 829)

cursor.limit()

`cursor.limit()`

Use the `cursor.limit()` (page 985) method on a cursor to specify the maximum number of documents the cursor will return. `cursor.limit()` (page 985) is analogous to the `LIMIT` statement in a SQL database.

Note: You must apply `cursor.limit()` (page 985) to the cursor before retrieving any documents from the database.

Use `cursor.limit()` (page 985) to maximize performance and prevent MongoDB from returning more results than required for processing.

A `cursor.limit()` (page 985) value of 0 (e.g. "`.limit(0)` (page 985)") is equivalent to setting no limit.

cursor.map()

`cursor.map(function)`

Applies `function` to each document visited by the cursor and collects the return values from successive application into an array.

The `cursor.map()` (page 986) method has the following parameter:

param function function A function to apply to each document visited by the cursor.

Example

```
db.users.find().map( function(u) { return u.name; } );
```

See also:

`cursor.forEach()` (page 984) for similar functionality.

cursor.max()

Definition

`cursor.max()`

Specifies the *exclusive* upper bound for a specific index in order to constrain the results of `find()` (page 951). `max()` (page 986) provides a way to specify an upper bound on compound key indexes.

The `max()` (page 986) method has the following parameter:

param document indexBounds The exclusive upper bound for the index keys.

The `indexBounds` parameter has the following prototype form:

```
{ field1: <max value>, field2: <max value2> ... fieldN:<max valueN>}
```

The fields correspond to *all* the keys of a particular index *in order*. You can explicitly specify the particular index with the `hint()` (page 985) method. Otherwise, `mongod` (page 1049) selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

See also:

`min()` (page 987).

Note: `max()` (page 986) is a shell wrapper around the query modifier `$max` (page 830).

Behavior

- Because `max()` (page 986) requires an index on a field, and forces the query to use this index, you may prefer the `$lt` (page 788) operator for the query if possible. Consider the following example:

```
db.products.find( { _id: 7 } ).max( { price: 1.39 } )
```

  The query will use the index on the `price` field, even if the index on `_id` may be better.

- `max()` (page 986) exists primarily to support the `mongos` (page 1061) (sharding) process.
- If you use `max()` (page 986) with `min()` (page 987) to specify a range, the index bounds specified in `min()` (page 987) and `max()` (page 986) must both refer to the keys of the same index.

Example This example assumes a collection named `products` that holds the following documents:

```
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
{ "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
{ "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
```

The collection has the following indexes:

```
{ "_id" : 1 }
{ "item" : 1, "type" : 1 }
{ "item" : 1, "type" : -1 }
{ "price" : 1 }
```

- Using the ordering of `{ item: 1, type: 1 }` index, `max()` (page 986) limits the query to the documents that are below the bound of `item` equal to `apple` and `type` equal to `jonagold`:

  ```
  db.products.find().max( { item: 'apple', type: 'jonagold' } ).hint( { item: 1, type: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
  { "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
  { "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
  ```

  If the query did not explicitly specify the index with the `hint()` (page 985) method, it is ambiguous as to whether `mongod` (page 1049) would select the `{ item: 1, type: 1 }` index ordering or the `{ item: 1, type: -1 }` index ordering.

- Using the ordering of the index `{ price: 1 }`, `max()` (page 986) limits the query to the documents that are below the index key bound of `price` equal to `1.99` and `min()` (page 987) limits the query to the documents that are at or above the index key bound of `price` equal to `1.39`:

  ```
  db.products.find().min( { price: 1.39 } ).max( { price: 1.99 } ).hint( { price: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
  { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
  { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
  { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
  { "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
  ```

cursor.min()

Definition

`cursor.`**`min`**`()`

Specifies the *inclusive* lower bound for a specific index in order to constrain the results of `find()` (page 951). `min()` (page 987) provides a way to specify lower bounds on compound key indexes.

The `min()` (page 987) has the following parameter:

param document indexBounds The inclusive lower bound for the index keys.

The `indexBounds` parameter has the following prototype form:

```
{ field1: <min value>, field2: <min value2>, fieldN:<min valueN> }
```

The fields correspond to *all* the keys of a particular index *in order*. You can explicitly specify the particular index with the `hint()` (page 985) method. Otherwise, MongoDB selects the index using the fields in the `indexBounds`; however, if multiple indexes exist on same fields with different sort orders, the selection of the index may be ambiguous.

See also:

`max()` (page 986).

Note: `min()` (page 987) is a shell wrapper around the query modifier `$min` (page 830).

Behaviors

- Because `min()` (page 987) requires an index on a field, and forces the query to use this index, you may prefer the `$gte` (page 787) operator for the query if possible. Consider the following example:

  ```
  db.products.find( { _id: 7 } ).min( { price: 1.39 } )
  ```

  The query will use the index on the `price` field, even if the index on `_id` may be better.

- `min()` (page 987) exists primarily to support the `mongos` (page 1061) process.

- If you use `min()` (page 987) with `max()` (page 986) to specify a range, the index bounds specified in `min()` (page 987) and `cursor.max()` must both refer to the keys of the same index.

Example This example assumes a collection named `products` that holds the following documents:

```
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
{ "_id" : 2, "item" : "apple", "type" : "fuji", "price" : 1.99 }
{ "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : 1.99 }
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
```

The collection has the following indexes:

```
{ "_id" : 1 }
{ "item" : 1, "type" : 1 }
{ "item" : 1, "type" : -1 }
{ "price" : 1 }
```

- Using the ordering of the `{ item: 1, type: 1 }` index, `min()` (page 987) limits the query to the documents that are at or above the index key bound of `item` equal to `apple` and `type` equal to `jonagold`, as in the following:

  ```
  db.products.find().min( { item: 'apple', type: 'jonagold' } ).hint( { item: 1, type: 1 } )
  ```

  The query returns the following documents:

  ```
  { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
  { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
  ```

```
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : 2.99 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : 1.99 }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : 0.99 }
```

If the query did not explicitly specify the index with the `hint()` (page 985) method, it is ambiguous as to whether `mongod` (page 1049) would select the `{ item: 1, type: 1 }` index ordering or the `{ item: 1, type: -1 }` index ordering.

- Using the ordering of the index `{ price: 1 }`, `min()` (page 987) limits the query to the documents that are at or above the index key bound of `price` equal to `1.39` and `max()` (page 986) limits the query to the documents that are below the index key bound of `price` equal to `1.99`:

```
db.products.find().min( { price: 1.39 } ).max( { price: 1.99 } ).hint( { price: 1 } )
```

The query returns the following documents:

```
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : 1.29 }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : 1.29 }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : 1.29 }
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : 1.29 }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : 1.39 }
```

cursor.next()

`cursor.next()`

Returns The next document in the cursor returned by the `db.collection.find()` (page 951) method. See `cursor.hasNext()` (page 985) related functionality.

cursor.objsLeftInBatch()

`cursor.objsLeftInBatch()`

`cursor.objsLeftInBatch()` (page 989) returns the number of documents remaining in the current batch.

The MongoDB instance returns response in batches. To retrieve all the documents from a cursor may require multiple batch responses from the MongoDB instance. When there are no more documents remaining in the current batch, the cursor will retrieve another batch to get more documents until the cursor exhausts.

cursor.readPref()

Definition

`cursor.readPref(`*mode*, *tagSet*`)`

Append `readPref()` (page 989) to a cursor to control how the client routes the query to members of the replica set.

param string mode One of the following *read preference* modes: `primary` (page 400), `primaryPreferred` (page 400), `secondary` (page 400), `secondaryPreferred` (page 400), or `nearest` (page 400)

param array tagSet A tag set used to specify custom read preference modes. For details, see *Tag Sets* (page 401).

Note: You must apply `cursor.readPref()` (page 989) 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 832) for related functionality.

cursor.size()

`cursor.size()`

Returns A count of the number of documents that match the `db.collection.find()` (page 951) query after applying any `cursor.skip()` (page 990) and `cursor.limit()` (page 985) methods.

cursor.skip()

`cursor.skip()`

Call the `cursor.skip()` (page 990) 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()` (page 990) 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()` (page 990) 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()` (page 990) will become slower and more CPU intensive. With larger collections, `cursor.skip()` (page 990) 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()` (page 990) method to a cursor to toggle the "snapshot" mode. This ensures that the query will not return a document multiple times, even if intervening write operations result in a move of the document due to the growth in document size.

> **Warning:**
> - You must apply `cursor.snapshot()` (page 990) to the cursor before retrieving any documents from the database.
> - You can only use `snapshot()` (page 990) with **unsharded** collections.

The `snapshot()` (page 990) does not guarantee isolation from insertion or deletions.

The `cursor.snapshot()` (page 990) traverses the index on the `_id` field. As such, `snapshot()` (page 990) **cannot** be used with `sort()` (page 991) or `hint()` (page 985).

Queries with results of less than 1 megabyte are effectively implicitly snapshotted.

cursor.sort()

Definition

`cursor.sort(sort)`

Controls 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()` (page 991) returns query results in ascending order for that attribute. If the field's corresponding value is negative, then `sort()` (page 991) returns query results in descending order.

The `sort()` (page 991) method has the following parameter:

param document sort A document that defines the sort order of the result set.

> **Note:** You must apply `cursor.limit()` (page 985) to the cursor before retrieving any documents from the database.

Example The following query returns all documents in `collection` sorted by the `age` field in descending order.

```
db.collection.find().sort( { age: -1 } );
```

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.

Limit Results Unless you have an index for the specified key pattern, use `cursor.sort()` (page 991) in conjunction with `cursor.limit()` (page 985) to avoid requiring MongoDB to perform a large, in-memory sort. `cursor.limit()` (page 985) 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()` (page 985), or create an index on the field that you're sorting to avoid this error.

Return Natural Order The `$natural` (page 833) 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.

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

cursor.toArray()

`cursor.toArray()`

The `toArray()` (page 992) method returns an array that contains all the documents from a cursor. The method iterates completely the cursor, loading all the documents into RAM and exhausting the cursor.

Returns An array of documents.

Consider the following example that applies `toArray()` (page 992) to the cursor returned from the `find()` (page 951) method:

```
var allProductsArray = db.products.find().toArray();

if (allProductsArray.length > 0) { printjson (allProductsArray[0]); }
```

The variable `allProductsArray` holds the array of documents returned by `toArray()` (page 992).

55.3.3 Database

Database Methods

Name	Description
`db.addUser()` (page 993)	Adds a user to a database, and allows administrators to configure the user
`db.auth()` (page 995)	Authenticates a user to a database.
`db.changeUserPassword()` (page 995)	Changes an existing user's password.

Table 55.2 – continued from previous page

Name	Description
db.cloneCollection() (page 996)	Copies data directly between MongoDB instances. Wraps cloneColle
db.cloneDatabase() (page 996)	Copies a database from a remote host to the current host. Wraps clone (
db.commandHelp() (page 996)	Returns help information for a *database command*.
db.copyDatabase() (page 996)	Copies a database to another database on the current host. Wraps copyd
db.createCollection() (page 997)	Creates a new collection. Commonly used to create a capped collection.
db.currentOp() (page 998)	Reports the current in-progress operations.
db.dropDatabase() (page 1002)	Removes the current database.
db.eval() (page 1002)	Passes a JavaScript function to the mongod (page 1049) instance for serv
db.fsyncLock() (page 1004)	Flushes writes to disk and locks the database to prevent write operations a
db.fsyncUnlock() (page 1004)	Allows writes to continue on a database locked with db.fsyncLock()
db.getCollection() (page 1005)	Returns a collection object. Used to access collections with names that ar
db.getCollectionNames() (page 1005)	Lists all collections in the current database.
db.getLastError() (page 1005)	Checks and returns the status of the last operation. Wraps getLastErr
db.getLastErrorObj() (page 1005)	Returns the status document for the last operation. Wraps getLastErr
db.getMongo() (page 1005)	Returns the Mongo() (page 1036) connection object for the current conn
db.getName() (page 1005)	Returns the name of the current database.
db.getPrevError() (page 1006)	Returns a status document containing all errors since the last error reset. W
db.getProfilingLevel() (page 1006)	Returns the current profiling level for database operations.
db.getProfilingStatus() (page 1006)	Returns a document that reflects the current profiling level and the profilin
db.getReplicationInfo() (page 1006)	Returns a document with replication statistics.
db.getSiblingDB() (page 1007)	Provides access to the specified database.
db.help() (page 1008)	Displays descriptions of common db object methods.
db.hostInfo() (page 1008)	Returns a document with information about the system MongoDB runs o
db.isMaster() (page 1008)	Returns a document that reports the state of the replica set.
db.killOp() (page 1009)	Terminates a specified operation.
db.listCommands() (page 1009)	Displays a list of common database commands.
db.loadServerScripts() (page 1009)	Loads all scripts in the system.js collection for the current database in
db.logout() (page 1009)	Ends an authenticated session.
db.printCollectionStats() (page 1010)	Prints statistics from every collection. Wraps db.collection.stats
db.printReplicationInfo() (page 1010)	Prints a report of the status of the replica set from the perspective of the p
db.printShardingStatus() (page 1010)	Prints a report of the sharding configuration and the chunk ranges.
db.printSlaveReplicationInfo() (page 1011)	Prints a report of the status of the replica set from the perspective of the s
db.removeUser() (page 1011)	Removes a user from a database.
db.repairDatabase() (page 1011)	Runs a repair routine on the current database.
db.resetError() (page 1011)	Resets the error message returned by db.getPrevError() (page 100
db.runCommand() (page 1012)	Runs a *database command* (page 833).
db.serverBuildInfo() (page 1012)	Returns a document that displays the compilation parameters for the mon
db.serverStatus() (page 1012)	Returns a document that provides an overview of the state of the database
db.setProfilingLevel() (page 1012)	Modifies the current level of database profiling.
db.shutdownServer() (page 1013)	Shuts down the current mongod (page 1049) or mongos (page 1061) pr
db.stats() (page 1013)	Returns a document that reports on the state of the current database.
db.version() (page 1013)	Returns the version of the mongod (page 1049) instance.

db.addUser()

Definition

`db.addUser(document)`

Use db.addUser() (page 993) to add privilege documents to the system.users (page 238) collection in a database, which creates database credentials in MongoDB.

Changed in version 2.4: The schema of `system.users` (page 238) changed in 2.4 to accommodate a more *sophisticated privilege model* (page 233). In 2.4 `db.addUser()` (page 993) supports both forms of privilege documents.

In MongoDB 2.4 you must pass `db.addUser()` (page 993) a document that contains a well-formed `system.users` (page 238) document. In MongoDB 2.2 pass arguments to `db.addUser()` (page 993) that describe *user credentials* (page 994). A 2.4 privilege document has a subset of the following fields:

field string user The username for a new database user.

field array roles An array of user roles.

field hash pwd A shared secret used to authenticate the user. The `pwd` field and the `userSource` field are mutually exclusive. The document cannot contain both.

field string userSource The database that contains the credentials for the user. The `userSource` field and the `pwd` field are mutually exclusive. The document cannot contain both.

field document otherDBRoles Roles this user has on other databases.

See *system.users Privilege Documents* (page 238) for documentation of the 2.4 privilege documents.

Examples The following are prototype `db.addUser()` (page 993) operations:

```
db.addUser( { user: "<user>", pwd: "<password>", roles: [<roles>] } )
```

This operation creates a `system.users` (page 238) document with a password using the `pwd` (page 239) field

In the following prototype, rather than specify a password directly, you can delegated the credential to another database using the `userSource` (page 239) field:

```
db.addUser( { user: "<user>", userSource: "<database>", roles: [<roles>] } )
```

To create and add a 2.4-style privilege document to `system.users` (page 238) to grant `readWrite` (page 234) privileges to a user named "author" with privileges, use the following operation:

```
db.addUser( { user: "author", pwd: "pass", roles: [ "readWrite" ] } )
```

If you want to store user credentials in a single `users` database, you can use *delegated credentials* (page 240), as in the following example:

```
db.addUser( { user: "author", userSource: "users", roles: [ "readWrite" ] } )
```

```
.. seealso::
```

- *Add a User to a Database* (page 226)
- *User Privilege Roles in MongoDB* (page 233)
- *system.users Privilege Documents* (page 238)

Legacy Privilege Documents To create legacy (2.2. and earlier) privilege documents, `db.addUser()` (page 993) accepts the following parameters:

param string user The username.

param string password The corresponding password.

param boolean readOnly Defaults to `false`. Grants users a restricted privilege set that only allows the user to read the this database.

The command takes the following form:

```
db.addUser( "<username>", "<password>",  <read-only> )
```

Example

To create and add a legacy (2.2. and earlier) privilege document with a user named `guest` and the password `pass` that has only `readOnly` privileges, use the following operation:

```
db.addUser( "guest", "pass", true )
```

Note: The `mongo` (page 1066) shell excludes all `db.addUser()` (page 993) operations from the saved history.

Deprecated since version 2.4: The `roles` parameter replaces the `readOnly` parameter for `db.addUser()` (page 993). 2.4 also adds the `otherDBRoles` (page 239) and `userSource` (page 239) fields to documents in the `system.users` (page 238) collection.

db.auth()

Definition

`db.auth(username, password)`

Allows a user to authenticate to the database from within the shell.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

Alternatively, you can use *mongo --username* and *--password* to specify authentication credentials.

Note: The `mongo` (page 1066) shell excludes all `db.auth()` (page 995) operations from the saved history.

db.changeUserPassword()

Definition

`db.changeUserPassword(username, password)`

Allows an administrator to update a user's password from within the shell.

param string username Specifies an existing username with access privileges for this database.

param string password Specifies the corresponding password.

Throws exception If an error occurs, the `changeUserPassword()` (page 995) helper throws an exception with the error message and code.

Example

Example

The following operation changes the `reporting` user's password to `SOhSS3TbYhxusooLiW8ypJPxmt1oOfL`:

```
db = db.getSiblingDB('records')
db.changeUserPassword("reporting", "SOhSS3TbYhxusooLiW8ypJPxmt1oOfL")
```

db.cloneCollection()

Definition

`db.cloneCollection(from, collection, query)`

Copies data directly between MongoDB instances. The `db.cloneCollection()` (page 996) wraps the `cloneCollection` (page 886) database command and accepts the following arguments:

param string from Host name of the MongoDB instance that holds the collection to copy.

param string collection The collection in the MongoDB instance that you want to copy. `db.cloneCollection()` (page 996) will only copy the collection with this name from *database* of the same name as the current database the remote MongoDB instance. If you want to copy a collection from a different database name you must use the `cloneCollection` (page 886) directly.

param document query A standard query document that limits the documents copied as part of the `db.cloneCollection()` (page 996) operation. All *query selectors* (page 785) available to the `find()` (page 951) are available here.

`db.cloneCollection()` (page 996) does not allow you to clone a collection through a `mongos` (page 1061). You must connect directly to the `mongod` (page 1049) instance.

db.cloneDatabase()

Definition

`db.cloneDatabase("hostname")`

Copies a remote database to the current database. The command assumes that the remote database has the same name as the current database.

param string hostname The hostname of the database to copy.

This method provides a wrapper around the MongoDB *database command* "`clone` (page 885)." The `copydb` (page 883) database command provides related functionality.

Example To clone a database named `importdb` on a host named `hostname`, issue the following:

```
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.

db.commandHelp()

Description

`db.commandHelp(command)`

Displays help text for the specified *database command*. See the *Database Commands* (page 833).

The `commandHelp()` (page 996) method has the following parameter:

param string command The name of a *database command*.

db.copyDatabase()

Definition

`db.copyDatabase(origin, destination, hostname)`

Copies a single logical *database* from a remote MongoDB instance to the local database. `db.copyDatabase()` (page 996) wraps the `copydb` (page 883) database command, and takes the following arguments:

param string origin The name of the database on the origin system.

param string destination The name of the database to copy the origin database into.

param string hostname The hostname of the origin database host. Omit the hostname to copy from one name to another on the same server.

`db.copyDatabase()` (page 996) implicitly creates the destination databases if it does not exist. If you do not specify the `hostname` argument, MongoDB assumes the origin and destination databases are on the *local* instance.

The `clone` (page 885) database command provides related functionality.

Example To copy a database named `records` into a database named `archive_records`, use the following invocation of `db.copyDatabase()` (page 996):

```
db.copyDatabase('records', 'archive_records')
```

db.createCollection()

Definition

`db.createCollection(name, options)`

Creates a new collection explicitly.

Because MongoDB creates a collection implicitly when the collection is first referenced in a command, this method is used primarily for creating new *capped collections*. This is also used to pre-allocate space for an ordinary collection.

The `createCollection` (page 997) has the following prototype form:

```
db.createCollection(name, {capped: <Boolean>, autoIndexID: <Boolean>, size: <number>, max <numbe
```

The `createCollection` (page 997) method has the following parameters:

param string name The name of the collection to create.

param document options Configuration options for creating a capped collection or for preallocating space in a new collection.

The `options` document creates a capped collection or preallocates space in a new ordinary collection. The `options` document contains the following fields:

field Boolean capped Enables a *capped collection*. To create a capped collection, specify `true`. If you specify `true`, you must also set a maximum size in the `size` field.

field Boolean autoIndexID If `capped` is `true`, specify `false` to disable the automatic creation of an index on the `_id` field. Before 2.2, the default value for `autoIndexId` was `false`. See *_id Fields and Indexes on Capped Collections* (page 1194) for more information.

field number size Specifies a maximum size in bytes for a capped collection. The `size` field is required for capped collections. If `capped` is false, you can use this field to preallocate space for an ordinary collection.

field number max The maximum number of documents allowed in the capped collection. The `size` limit takes precedence over this limit. If a capped collection reaches its maximum `size` before it reaches the maximum number of documents, MongoDB removes old documents. If you prefer to use this limit, ensure that the `size` limit, which is required, is sufficient to contain the documents limit.

Example The following example creates a capped 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 and may also specify a maximum document count. MongoDB removes 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 : 5242880, max : 5000 } )
```

This command creates a collection named `log` with a maximum size of 5 megabytes and 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` (page 885). See *Capped Collections* (page 578) for more information about capped collections.

db.currentOp()

Definition

`db.currentOp()`

Returns A *document* that reports in-progress operations for the database instance.

The `db.currentOp()` (page 998) method can take no arguments or take the `true` argument, which returns a more verbose output, including idle connections and system operations. The following example uses the `true` argument:

```
db.currentOp(true)
```

`db.currentOp()` (page 998) is available only for users with administrative privileges.

You can use `db.killOp()` (page 1009) in conjunction with the `opid` (page 1000) field to terminate a currently running operation. The following JavaScript operations for the `mongo` (page 1066) shell filter the output of 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)
     })
```

> **Warning:** Terminate running operations with extreme caution. Only use `db.killOp()` (page 1009) to terminate operations initiated by clients and *do not* terminate internal database operations.

Example The following is an example of `db.currentOp()` (page 998) output. If you specify the `true` argument, `db.currentOp()` (page 998) returns more verbose 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()
                   }
               }
             },
            ]
}
```

Output Changed in version 2.2.

The db.currentOp() (page 998) returns a document with an array named inprog. The inprog array contains a document for each in-progress operation. The fields that appear for a given operation depend on the kind of operation and its state.

currentOp.**opid**
Holds an identifier for the operation. You can pass this value to db.killOp() (page 1009) in the mongo (page 1066) shell to terminate the operation.

currentOp.**active**
A boolean value, that is true if the operation has started or false if the operation is queued and waiting for a lock to run. active (page 1000) may be true even if the operation has yielded to another operation.

currentOp.**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.

If the operation is not running, (i.e. if active (page 1000) is false,) this field may not appear in the output of db.currentOp() (page 998).

currentOp.**op**
A string that identifies the type of operation. The possible values are:

- insert
- query
- update
- remove
- getmore
- command

currentOp.**ns**
The *namespace* the operation targets. MongoDB forms namespaces using the name of the *database* and the name of the *collection*.

currentOp.**query**
A document containing the current operation's query. The document is empty for operations that do not have queries: getmore, insert, and command.

currentOp.**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 (page 851) and db.eval() (page 1002), the client will be 0.0.0.0:0, rather than an actual client.

currentOp.**desc**
A description of the client. This string includes the connectionId (page 1000).

currentOp.**threadId**
An identifier for the thread that services the operation and its connection.

currentOp.**connectionId**
An identifier for the connection where the operation originated.

currentOp.**locks**
New in version 2.2.

The `locks` (page 1000) document reports on the kinds of locks the operation currently holds. The following kinds of locks are possible:

`currentOp.locks.`**^**

`^` (page 1001) reports on the use of the global lock for the `mongod` (page 1049) instance. All operations must hold the global lock for some phases of operation.

`currentOp.locks.`**^local**

`^local` (page 1001) 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.

`currentOp.locks.`**^<database>**

`locks.^<database>` (page 1001) reports on the lock state for the database that this operation targets.

`locks` (page 1000) replaces `lockType` in earlier versions.

`currentOp.`**lockType**

Changed in version 2.2: The `locks` (page 1000) replaced the `lockType` (page 1001) field in 2.2.

Identifies the type of lock the operation currently holds. The possible values are:

- `read`
- `write`

`currentOp.`**waitingForLock**

Returns a boolean value. `waitingForLock` (page 1001) is `true` if the operation is waiting for a lock and `false` if the operation has the required lock.

`currentOp.`**msg**

The `msg` (page 1001) 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.

`currentOp.`**progress**

Reports on the progress of mapReduce or indexing operations. The `progress` (page 1001) fields corresponds to the completion percentage in the `msg` (page 1001) field. The `progress` (page 1001) specifies the following information:

`currentOp.progress.`**done**

Reports the number completed.

`currentOp.progress.`**total**

Reports the total number.

`currentOp.`**killed**

Returns `true` if `mongod` (page 1049) instance is in the process of killing the operation.

`currentOp.`**numYields**

`numYields` (page 1001) 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.

`currentOp.`**lockStats**

New in version 2.2.

The `lockStats` (page 1001) document reflects the amount of time the operation has spent both acquiring and holding locks. `lockStats` (page 1001) 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.

`currentOp.`**`timeLockedMicros`**

The `timeLockedMicros` (page 1002) document reports the amount of time the operation has spent holding a specific lock.

For operations that require more than one lock, like those that lock the `local` database to update the *oplog*, then the values in this document can be longer than this value may be longer than the total length of the operation (i.e. `secs_running` (page 1000).)

`currentOp.timeLockedMicros.`**`R`**

Reports the amount of time in microseconds the operation has held the global read lock.

`currentOp.timeLockedMicros.`**`W`**

Reports the amount of time in microseconds the operation has held the global write lock.

`currentOp.timeLockedMicros.`**`r`**

Reports the amount of time in microseconds the operation has held the database specific read lock.

`currentOp.timeLockedMicros.`**`w`**

Reports the amount of time in microseconds the operation has held the database specific write lock.

`currentOp.`**`timeAcquiringMicros`**

The `timeAcquiringMicros` (page 1002) document reports the amount of time the operation has spent *waiting* to acquire a specific lock.

`currentOp.timeAcquiringMicros.`**`R`**

Reports the mount of time in microseconds the operation has waited for the global read lock.

`currentOp.timeAcquiringMicros.`**`W`**

Reports the mount of time in microseconds the operation has waited for the global write lock.

`currentOp.timeAcquiringMicros.`**`r`**

Reports the mount of time in microseconds the operation has waited for the database specific read lock.

`currentOp.timeAcquiringMicros.`**`w`**

Reports the mount of time in microseconds the operation has waited for the database specific write lock.

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()

Definition

db.**eval** (*function*, *arguments*)

Provides the ability to run JavaScript code on the MongoDB server.

The helper db.eval() (page 1002) in the mongo (page 1066) shell wraps the eval (page 862) command. Therefore, the helper method shares the characteristics and behavior of the underlying command with *one exception*: db.eval() (page 1002) method does not support the nolock option.

The method accepts the following parameters:

param JavaScript function function A JavaScript function to execute.

param list arguments A list of arguments to pass to the JavaScript function. Omit if the function does not take arguments.

The JavaScript function need not take any arguments, as in the first example, or may optionally take arguments as in the second:

```
function () {
  // ...
}

function (arg1, arg2) {
   // ...
}
```

Examples The following is an example of the db.eval() (page 1002) method:

```
db.eval( function(name, incAmount) {
            var doc = db.myCollection.findOne( { name : name } );

            doc = doc || { name : name , num : 0 , total : 0 , avg : 0 };

            doc.num++;
            doc.total += incAmount;
            doc.avg = doc.total / doc.num;

            db.myCollection.save( doc );
            return doc;
         },
         "eliot", 5 );
```

- The db in the function refers to the current database.
- "eliot" is the argument passed to the function, and corresponds to the name argument.
- 5 is an argument to the function and corresponds to the incAmount field.

If you want to use the server's interpreter, you must run db.eval() (page 1002). Otherwise, the mongo (page 1066) shell's JavaScript interpreter evaluates functions entered directly into the shell.

If an error occurs, db.eval() (page 1002) throws an exception. The following is an example of an invalid function that uses the variable x without declaring it as an argument:

```
db.eval( function() { return x + x; }, 3 );
```

The statement results in the following exception:

```
{
   "errmsg" : "exception: JavaScript execution failed: ReferenceError: x is not defined near '{ retur
   "code" : 16722,
```

```
    "ok" : 0
}
```

Warning:

- By default, `db.eval()` (page 1002) takes a global write lock before evaluating the JavaScript function. As a result, `db.eval()` (page 1002) blocks all other read and write operations to the database while the `db.eval()` (page 1002) operation runs. Set `nolock` to `true` on the `eval` (page 862) *command* to prevent the `eval` (page 862) *command* from taking the global write lock before evaluating the JavaScript. `nolock` does not impact whether operations within the JavaScript code itself takes a write lock.
- Do not use `db.eval()` (page 1002) for long running operations as `db.eval()` (page 1002) blocks all other operations. Consider using *other server side code execution options* (page 581).
- You can not use `db.eval()` (page 1002) with *sharded* data. In general, you should avoid using `db.eval()` (page 1002) in *sharded cluster*; nevertheless, it is possible to use `db.eval()` (page 1002) with non-sharded collections and databases stored in a *sharded cluster*.
- With `authentication` (page 1118) enabled, `db.eval()` (page 1002) will fail during the operation if you do not have the permission to perform a specified task.
  Changed in version 2.4: You must have full admin access to run.

Changed in version 2.4: The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, `db.eval()` (page 1002) executed in a single thread.

See also:

Server-side JavaScript (page 581)

db.fsyncLock()

db.**fsyncLock**()

Forces the `mongod` (page 1049) to flush pending all write operations to the disk and locks the *entire* `mongod` (page 1049) instance to prevent additional writes until the user releases the lock with the `db.fsyncUnlock()` (page 1004) command. `db.fsyncLock()` (page 1004) is an administrative command.

This command provides a simple wrapper around a `fsync` (page 888) database command with the following syntax:

```
{ fsync: 1, lock: true }
```

This function locks the database and create a window for *backup operations* (page 133).

Note: The database cannot be locked with `db.fsyncLock()` (page 1004) while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()` (page 1004). Disable profiling using `db.setProfilingLevel()` (page 1012) as follows in the `mongo` (page 1066) shell:

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

db.fsyncUnlock()

db.**fsyncUnlock**()

Unlocks a `mongod` (page 1049) instance to allow writes and reverses the operation of a `db.fsyncLock()` (page 1004) operation. Typically you will use `db.fsyncUnlock()` (page 1004) following a database *backup operation* (page 133).

`db.fsyncUnlock()` (page 1004) is an administrative command.

db.getCollection()

Description

`db.getCollection(name)`

Returns a collection name. This is useful for a collection whose name might interact with the shell itself, such names that begin with _ or that mirror the *database commands* (page 833).

The `getCollection()` (page 1005) method has the following parameter:

param string name The name of the collection.

db.getCollectionNames()

`db.getCollectionNames()`

Returns An array containing all collections in the existing database.

db.getLastError()

`db.getLastError()`

Returns The last error message string.

Sets the level of *write concern* for confirming the success of write operations.

See

`getLastError` (page 861) for all options, *Write Concern* (page 395) for a conceptual overview, *Write Operations* (page 53) for information about all write operations in MongoDB, and *Replica Set Write Concern* (page 395) for special considerations related to write concern for replica sets.

db.getLastErrorObj()

`db.getLastErrorObj()`

Returns A full *document* with status information.

db.getMongo()

`db.getMongo()`

Returns The current database connection.

`db.getMongo()` (page 1005) runs when the shell initiates. Use this command to test that the `mongo` (page 1066) 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` (page 862) (also `db.resetError()` (page 1011)) command.

This method provides a wrapper around the `getPrevError` (page 862) command.

db.getProfilingLevel()

`db.getProfilingLevel()`

This method provides a wrapper around the database command "`profile` (page 907)" and returns the current profiling level.

Deprecated since version 1.8.4: Use `db.getProfilingStatus()` (page 1006) for related functionality.

db.getProfilingStatus()

`db.getProfilingStatus()`

Returns The current `profile` (page 907) level and `slowms` (page 1121) setting.

db.getReplicationInfo()

Definition

`db.getReplicationInfo()`

Returns A document with the status of the replica status, using data polled from the "*oplog*". Use this output when diagnosing issues with replication.

Output

`db.getReplicationInfo.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.

`db.getReplicationInfo.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.

`db.getReplicationInfo.errmsg`

Returns an error message if there are no entries in the oplog.

`db.getReplicationInfo.oplogMainRowCount`

Only present when there are no entries in the oplog. Reports a the number of items or rows in the *oplog* (e.g. `0`).

`db.getReplicationInfo.timeDiff`

Returns the difference between the first and last operation in the *oplog*, represented in seconds.

Only present if there are entries in the oplog.

db.getReplicationInfo.**timeDiffHours**

Returns the difference between the first and last operation in the *oplog*, rounded and represented in hours.

Only present if there are entries in the oplog.

db.getReplicationInfo.**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.

Only present if there are entries in the oplog.

db.getReplicationInfo.**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.

Only present if there are entries in the oplog.

db.getReplicationInfo.**now**

Returns a time stamp that reflects 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()`.

Only present if there are entries in the oplog.

db.getSiblingDB()

Definition

db.**getSiblingDB**(<*database*>)

param string database The name of a MongoDB database.

Returns A database object.

Used to return another database without modifying the `db` variable in the shell environment.

Example You can use `db.getSiblingDB()` (page 1007) as an alternative to the `use <database>` helper. This is particularly useful when writing scripts using the `mongo` (page 1066) shell where the `use` helper is not available. Consider the following sequence of operations:

```
db = db.getSiblingDB('users')
db.active.count()
```

This operation sets the `db` object to point to the database named `users`, and then returns a *count* (page 946) of the collection named `active`. You can create multiple `db` objects, that refer to different databases, as in the following sequence of operations:

```
users = db.getSiblingDB('users')
records = db.getSiblingDB('records')

users.active.count()
users.active.findOne()

records.requests.count()
records.requests.findOne()
```

This operation creates two `db` objects referring to different databases (i.e. `users` and `records`,) and then returns a *count* (page 946) and an *example document* (page 955) from one collection in that database (i.e. `active` and `requests` respectively.)

db.help()

db.**help**()

Returns Text output listing common methods on the db object.

db.hostInfo()

db.**hostInfo**()

New in version 2.2.

Returns A document with information about the underlying system that the mongod (page 1049) or mongos (page 1061) runs on. Some of the returned fields are only included on some platforms.

db.hostInfo() (page 1008) provides a helper in the mongo (page 1066) shell around the hostInfo (page 917) The output of db.hostInfo() (page 1008) on a Linux system will resemble the following:

```
{
   "system" : {
          "currentTime" : ISODate("<timestamp>"),
          "hostname" : "<hostname>",
          "cpuAddrSize" : <number>,
          "memSizeMB" : <number>,
          "numCores" : <number>,
          "cpuArch" : "<identifier>",
          "numaEnabled" : <boolean>
   },
   "os" : {
          "type" : "<string>",
          "name" : "<string>",
          "version" : "<string>"
   },
   "extra" : {
          "versionString" : "<string>",
          "libcVersion" : "<string>",
          "kernelVersion" : "<string>",
          "cpuFrequencyMHz" : "<string>",
          "cpuFeatures" : "<string>",
          "pageSize" : <number>,
          "numPages" : <number>,
          "maxOpenFiles" : <number>
   },
   "ok" : <return>
}
```

See hostInfo (page 918) for full documentation of the output of db.hostInfo() (page 1008).

db.isMaster()

db.**isMaster**()

Returns A document that describes the role of the mongod (page 1049) instance.

If the mongod (page 1049) is a member of a *replica set*, then the ismaster (page 871) and secondary (page 872) fields report if the instance is the *primary* or if it is a *secondary* member of the replica set.

See

isMaster (page 871) for the complete documentation of the output of isMaster() (page 1008).

db.killOp()

Description

`db.killOp(opid)`

Terminates an operation as specified by the operation ID. To find operations and their corresponding IDs, see db.currentOp() (page 998).

The killOp() (page 1009) method has the following parameter:

param number opid An operation ID.

> **Warning:** Terminate running operations with extreme caution. Only use db.killOp() (page 1009) to terminate operations initiated by clients and *do not* terminate internal database operations.

db.listCommands()

`db.listCommands()`

Provides a list of all database commands. See the "*Database Commands* (page 833)" document for a more extensive index of these options.

db.loadServerScripts()

`db.loadServerScripts()`

db.loadServerScripts() (page 1009) loads all scripts in the system.js collection for the current database into the mongo (page 1066) 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 (page 797) clauses and mapReduce (page 840) operations.

db.logout()

`db.logout()`

Ends the current authentication session. This function has no effect if the current session is not authenticated.

Note: If you're not logged in and using authentication, db.logout() (page 1009) has no effect.

Changed in version 2.4: Because MongoDB now allows users defined in one database to have privileges on another database, you must call db.logout() (page 1009) while using the same database context that you authenticated to.

If you authenticated to a database such as users or $external, you must issue db.logout() (page 1009) against this database in order to successfully log out.

Example

Use the `use <database-name>` helper in the interactive `mongo` (page 1066) shell, or the following `db.getSiblingDB()` (page 1007) in the interactive shell or in `mongo` (page 1066) shell scripts to change the `db` object:

```
db = db.getSiblingDB('<database-name>')
```

When you have set the database context and `db` object, you can use the `db.logout()` (page 1009) to log out of database as in the following operation:

```
db.logout()
```

`db.logout()` (page 1009) function provides a wrapper around the database command `logout`.

db.printCollectionStats()

`db.printCollectionStats()`
Provides a wrapper around the `db.collection.stats()` (page 973) method. Returns statistics from every collection separated by three hyphen characters.

Note: The `db.printCollectionStats()` (page 1010) in the `mongo` (page 1066) shell does **not** return *JSON*. Use `db.printCollectionStats()` (page 1010) for manual inspection, and `db.collection.stats()` (page 973) in scripts.

See also:

collStats (page 900)

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 *replSetGetStatus* (page 865) for more information regarding the contents of this output.

Note: The `db.printReplicationInfo()` (page 1010) in the `mongo` (page 1066) shell does **not** return *JSON*. Use `db.printReplicationInfo()` (page 1010) for manual inspection, and `rs.status()` (page 1017) in scripts.

db.printShardingStatus()

Definition

`db.printShardingStatus()`
Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*.

Only use `db.printShardingStatus()` (page 1010) when connected to a `mongos` (page 1061) instance.

The `db.printShardingStatus()` (page 1010) method has the following parameter:

param Boolean verbose If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See *sh.status()* (page 1029) for details of the output.

Note: The `db.printShardingStatus()` (page 1010) in the `mongo` (page 1066) shell does **not** return *JSON*. Use `db.printShardingStatus()` (page 1010) for manual inspection, and *Config Database* (page 555) in scripts.

See also:

`sh.status()` (page 1029)

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 *replSetGetStatus* (page 865) for more information regarding the contents of this output.

Note: The `db.printSlaveReplicationInfo()` (page 1011) in the `mongo` (page 1066) shell does **not** return *JSON*. Use `db.printSlaveReplicationInfo()` (page 1011) for manual inspection, and `rs.status()` (page 1017) in scripts.

---

db.removeUser()

Definition

`db.removeUser(username)`

Removes the specified username from the database.

The `db.removeUser()` (page 1011) method has the following parameter:

param string username The database username.

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` (page 895) or related options like `db.repairDatabase()` (page 1011) in the `mongo` (page 1066) shell or *mongod --repair*. Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run `repairDatabase` (page 895). In the event of an unclean shutdown, the server will be able restore the data files to a pristine state automatically.

`db.repairDatabase()` (page 1011) provides a wrapper around the database command `repairDatabase` (page 895), and has the same effect as the run-time option *mongod --repair* option, limited to *only* the current database. See `repairDatabase` (page 895) for full documentation.

db.resetError()

`db.resetError()`

Deprecated since version 1.6.

Resets the error message returned by `db.getPrevError` (page 1006) or `getPrevError` (page 862). Provides a wrapper around the `resetError` (page 862) command.

db.runCommand()

Definition

`db.runCommand(command)`

Provides a helper to run specified *database commands* (page 833). This is the preferred method to issue database commands, as it provides a consistent interface between the shell and drivers.

param document,string command "A *database command*, specified either in *document* form or as a string. If specified as a string, `db.runCommand()` (page 1012) transforms the string into a document."

db.serverBuildInfo()

`db.serverBuildInfo()`

Provides a wrapper around the `buildInfo` (page 899) *database command*. `buildInfo` (page 899) returns a document that contains an overview of parameters used to compile this `mongod` (page 1049) instance.

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` (page 919).

Changed in version 2.4: In 2.4 you can dynamically suppress portions of the `db.serverStatus()` (page 1012) output, or include suppressed sections in a document passed to the `db.serverStatus()` (page 1012) method, as in the following example:

```
db.serverStatus( { repl: 0, indexCounters: 0, locks: 0 } )
db.serverStatus( { workingSet: 1, metrics: 0, locks: 0 } )
```

`db.serverStatus()` (page 1012) includes all fields by default, except `workingSet` (page 932), by default.

Note: You may only dynamically include top-level fields from the *serverStatus* (page 919) document that are not included by default. You can exclude any field that `db.serverStatus()` (page 1012) includes by default.

See also:

"*serverStatus* (page 919)" for complete documentation of the output of this function.

db.setProfilingLevel()

Definition

`db.setProfilingLevel(level, slowms)`

Modifies the current *database profiler* level used by the database profiling system to capture data about performance. The method provides a wrapper around the *database command* `profile` (page 907).

param integer level Specifies a profiling level, which is either `0` for no profiling, `1` for only slow operations, or `2` for all operations.

param integer slowms Sets the threshold in milliseconds for the profile to consider a query or operation to be slow.

The level chosen can affect performance. It also can allow the server to write the contents of queries to the log, which might have information security implications for your deployment.

Configure the `slowms` (page 1121) option to set the threshold for the profiler to consider a query "slow." Specify this value in milliseconds to override the default.

`mongod` (page 1049) writes the output of the database profiler to the `system.profile` collection.

`mongod` (page 1049) prints information about queries that take longer than the `slowms` (page 1121) to the log even when the database profiler is not active.

Note: The database cannot be locked with `db.fsyncLock()` (page 1004) while profiling is enabled. You must disable profiling before locking the database with `db.fsyncLock()` (page 1004). Disable profiling using `db.setProfilingLevel()` (page 1012) as follows in the `mongo` (page 1066) shell:

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

db.shutdownServer()

`db.shutdownServer()`

Shuts down the current `mongod` (page 1049) or `mongos` (page 1061) process cleanly and safely.

This operation fails when the current database *is not* the *admin database*.

This command provides a wrapper around the `shutdown` (page 896).

db.stats()

Description

`db.stats(scale)`

Returns statistics that reflect the use state of a single *database*.

The `db.stats()` (page 1013) method has the following parameter:

param number scale The scale at which to deliver results. Unless specified, this command returns all data in bytes.

Returns A *document* with statistics reflecting the database system's state. For an explanation of the output, see *dbStats* (page 904).

The `db.stats()` (page 1013) method is a wrapper around the `dbStats` (page 904) database command.

Example The following example converts the returned values to kilobytes:

```
db.stats(1024)
```

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` (page 1049) or `mongos` (page 1061) instance.

55.3.4 Replication

Replication Methods

Name	Description
rs.add() (page 1014)	Adds a member to a replica set.
rs.addArb() (page 1015)	Adds an *arbiter* to a replica set.
rs.conf() (page 1015)	Returns the replica set configuration document.
rs.freeze() (page 1015)	Prevents the current member from seeking election as primary for a period of time.
rs.help() (page 1016)	Returns basic help text for *replica set* functions.
rs.initiate() (page 1016)	Initializes a new replica set.
rs.reconfig() (page 1016)	Re-configures a replica set by applying a new replica set configuration object.
rs.remove() (page 1017)	Remove a member from a replica set.
rs.slaveOk() (page 1017)	Sets the slaveOk property for the current connection. Deprecated. Use readPref() (page 989) and Mongo.setReadPref() (page 1035) to set *read preference*.
rs.status() (page 1017)	Returns a document with information about the state of the replica set.
rs.stepDown() (page 1018)	Causes the current *primary* to become a secondary which forces an *election*.
rs.syncFrom() (page 1018)	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

rs.add()

Definition

rs.**add**(*host*, *arbiterOnly*)

Adds a member to a *replica set*.

param string,document host The new member to add to the replica set. If a string, specifies the hostname and optionally the port number for the new member. If a document, specifies a replica set members document, as found in the members (page 474) array. To view a replica set's members array, run rs.conf() (page 1015).

param boolean arbiterOnly Applies only if the <host> value is a string. If true, the added host is an arbiter."

You may specify new hosts in one of two ways:

1. as a "hostname" with an optional port number to use the default configuration as in the *Add a Member to an Existing Replica Set* (page 429) example.
2. as a configuration *document*, as in the *Configure and Add a Member* (page 429) example.

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

rs.add() (page 1014) provides a wrapper around some of the functionality of the "replSetReconfig (page 868)" *database command* and the corresponding shell helper rs.reconfig() (page 1016). See the

Replica Set Configuration (page 473) document for full documentation of all replica set configuration options.

Example To add a `mongod` (page 1049) accessible on the default port `27017` running on the host `mongodb3.example.net`, use the following `rs.add()` (page 1014) invocation:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

To add `mongodb3.example.net` as a *secondary-only* (page 378) member of set, use the following form of `rs.add()` (page 1014):

```
rs.add( { "_id": 3, "host": "mongodbd3.example.net:27017", "priority": 0 } )
```

Replace, `3` with the next unused `_id` value in the replica set. See `rs.conf()` (page 1015) to see the existing `_id` values in the replica set configuration document.

See the *Replica Set Configuration* (page 473) and *Replica Set Tutorials* (page 415) documents for more information.

rs.addArb()

Description

`rs.addArb(host)`

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

The `rs.addArb()` (page 1015) method takes the following parameter:

param string host Specifies the hostname and optionally the port number of the arbiter member to add to replica set.

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

rs.conf()

`rs.conf()`

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

`rs.config()`

`rs.config()` (page 1015) is an alias of `rs.conf()` (page 1015).

rs.freeze()

Description

`rs.freeze(seconds)`

Makes the current *replica set* member ineligible to become *primary* for the period specified.

The `rs.freeze()` (page 1015) method has the following parameter:

param number seconds The duration the member is ineligible to become primary.

`rs.freeze()` (page 1015) provides a wrapper around the *database command* `replSetFreeze` (page 865).

rs.help()

rs.**help**()

Returns a basic help text for all of the *replication* (page 367) related shell functions.

rs.initiate()

Description

rs.**initiate**(*configuration*)

Initiates a *replica set*. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set.

The `rs.initiate()` (page 1016) method has the following parameter:

param document configuration A *document* that specifies *configuration settings* (page 473) for tne new replica set. If a configuration is not specified, MongoDB uses a default configuration.

The `rs.initiate()` (page 1016) method provides a wrapper around the "`replSetInitiate` (page 866)" *database command*.

Replica Set Configuration See *Member Configuration Tutorials* (page 431) and *Replica Set Configuration* (page 473) for examples of replica set configuration and invitation objects.

rs.reconfig()

Definition

rs.**reconfig**(*configuration*, *force*)

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

param document configuration A *document* that specifies the configuration of a replica set.

param document force "If set as `{ force: true }`, this forces the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to term:*rollback* situations."

`rs.reconfig()` (page 1016) overwrites the existing replica set configuration. Retrieve the current configuration object with `rs.conf()` (page 1015), modify the configuration as needed and then use `rs.reconfig()` (page 1016) to submit the modified configuration object.

`rs.reconfig()` (page 1016) provides a wrapper around the "`replSetReconfig` (page 868)" *database command*.

Examples To reconfigure a replica set, use the following sequence of operations:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf)
```

If you want to force the reconfiguration if a majority of the set is not connected to the current member, or you are issuing the command against a secondary, use the following form:

```
conf = rs.conf()

// modify conf to change configuration

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

Warning: Forcing a `rs.reconfig()` (page 1016) can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

See also:

Replica Set Configuration (page 473) and *Replica Set Tutorials* (page 415).

rs.remove()

Definition

`rs.remove(hostname)`

Removes the member described by the `hostname` parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which member will be *primary*. As a result, the shell will display an error even if this command succeeds.

The `rs.remove()` (page 1017) method has the following parameter:

param string hostname The hostname of a system in the replica set.

Note: Before running the `rs.remove()` (page 1017) 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()` (page 1017), 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* members. See the `readPref()` (page 989) method for more fine-grained control over *read preference* (page 398) in the `mongo` (page 1066) 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` (page 865) *database command*.

rs.stepDown()

Description

`rs.stepDown(seconds)`

Forces the current *replica set* member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current member is not the primary.

The `rs.stepDown()` (page 1018) method has the following parameter:

param number seconds The duration of time that the stepped-down member attempts to avoid re-election as primary. If this parameter is not specified, the method uses the default value of 60 seconds.

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

`rs.stepDown()` (page 1018) provides a wrapper around the *database command* `replSetStepDown` (page 868).

rs.syncFrom()

`rs.syncFrom()`

New in version 2.2.

Provides a wrapper around the `replSetSyncFrom` (page 869), which allows administrators to configure the member of a replica set that the current member will pull data from. Specify the name of the member you want to replicate from in the form of `[hostname]:[port]`.

See `replSetSyncFrom` (page 869) for more details.

55.3.5 Sharding

Sharding Methods

Name	Description
sh._adminCommand (page 1021)	Runs a *database command* against the admin database, like db.runCommand() (page 1012), but can confirm that it is issued against a mongos (page 1061).
sh._checkFullName() (page 1021)	Tests a namespace to determine if its well formed.
sh._checkMongos() (page 1021)	Tests to see if the mongo (page 1066) shell is connected to a mongos (page 1061) instance.
sh._lastMigration() (page 1021)	Reports on the last *chunk* migration.
sh.addShard() (page 1022)	Adds a *shard* to a sharded cluster.
sh.addShardTag() (page 1023)	Associates a shard with a tag, to support *tag aware sharding* (page 547).
sh.addTagRange() (page 1023)	Associates range of shard keys with a shard tag, to support *tag aware sharding* (page 547).
sh.disableBalancing() (page 1024)	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
sh.enableBalancing() (page 1024)	Activates the sharded collection balancer process if previously disabled using sh.disableBalancing() (page 1024).
sh.enableSharding() (page 1024)	Enables sharding on a specific database.
sh.getBalancerHost() (page 1025)	Returns the name of a mongos (page 1061) that's responsible for the balancer process.
sh.getBalancerState() (page 1025)	Returns a boolean to report if the *balancer* is currently enabled.
sh.help() (page 1025)	Returns help text for the sh methods.
sh.isBalancerRunning() (page 1026)	Returns a boolean to report if the balancer process is currently migrating chunks.
sh.moveChunk() (page 1026)	Migrates a *chunk* in a *sharded cluster*.
sh.removeShardTag() (page 1027)	Removes the association between a shard and a shard tag shard tag.
sh.setBalancerState() (page 1027)	Enables or disables the *balancer* which migrates *chunks* between *shards*.
sh.shardCollection() (page 1027)	Enables sharding for a collection.
sh.splitAt() (page 1028)	Divides an existing *chunk* into two chunks using a specific value of the *shard key* as the dividing point.
sh.splitFind() (page 1028)	Divides an existing *chunk* that contains a document matching a query into two approximately equal chunks.
sh.startBalancer() (page 1029)	Enables the *balancer* and waits for balancing to start.
sh.status() (page 1029)	Reports on the status of a *sharded cluster*, as db.printShardingStatus() (page 1010).
sh.stopBalancer() (page 1031)	Disables the *balancer* and waits for any in progress balancing rounds to complete.
sh.waitForBalancer() (page 1032)	Internal. Waits for the balancer state to change.
sh.waitForBalancerOff() (page 1032)	Internal. Waits until the balancer stops running.
sh.waitForDLock() (page 1033)	Internal. Waits for a specified distributed *sharded cluster* lock.
sh.waitForPingChange() (page 1033)	Internal. Waits for a change in ping state from one of the mongos (page 1061) in the sharded cluster.

sh._adminCommand()

Definition

`sh._adminCommand(command, checkMongos)`

Runs a database command against the admin database of a `mongos` (page 1061) instance.

param string command A database command to run against the `admin` database.

param boolean checkMongos Require verification that the shell is connected to a `mongos` (page 1061) instance.

See also:

`db.runCommand()` (page 1012)

sh._checkFullName()

Definition

`sh._checkFullName(namespace)`

Verifies that a *namespace* name is well formed. If the namespace is well formed, the `sh._checkFullName()` (page 1021) method exits *with no message*.

Throws If the namespace is not well formed, `sh._checkFullName()` (page 1021) throws: "name needs to be fully qualified <db>.<collection>"

The `sh._checkFullName()` (page 1021) method has the following parameter:

param string namespace The *namespace* of a collection. The namespace is the combination of the database name and the collection name. Enclose the namespace in quotation marks.

sh._checkMongos()

`sh._checkMongos()`

Returns nothing

Throws "not connected to a mongos"

The `sh._checkMongos()` (page 1021) method throws an error message if the `mongo` (page 1066) shell is not connected to a `mongos` (page 1061) instance. Otherwise it exits (no return document or return code).

sh._lastMigration()

Definition

`sh._lastMigration(namespace)`

Returns information on the last migration performed on the specified database or collection.

The `sh._lastMigration()` (page 1021) method has the following parameter:

param string namespace The *namespace* of a database or collection within the current database.

Output The `sh._lastMigration()` (page 1021) method returns a document with details about the last migration performed on the database or collection. The document contains the following output:

`sh._lastMigration._id`

The id of the migration task.

sh._lastMigration.**server**
The name of the server.

sh._lastMigration.**clientAddr**
The IP address and port number of the server.

sh._lastMigration.**time**
The time of the last migration, formatted as *ISODate*.

sh._lastMigration.**what**
The specific type of migration.

sh._lastMigration.**ns**
The complete *namespace* of the collection affected by the migration.

sh._lastMigration.**details**
A document containing details about the migrated chunk. The document includes `min` and `max` sub-documents with the bounds of the migrated chunk.

sh.addShard()

Definition

sh.**addShard**(*host*)
Adds a database instance or replica set to a *sharded cluster*. The optimal configuration is to deploy shards across *replica sets*. This method must be run on a `mongos` (page 1061) instance.

The `sh.addShard()` (page 1022) method has the following parameter:

param string host The hostname of either a standalone database instance or of a replica set. Include the port number if the instance is running on a non-standard port. Include the replica set name if the instance is a replica set, as explained below.

The `sh.addShard()` (page 1022) method has the following prototype form:

```
sh.addShard("<host>")
```

The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[replica-set-name]/[hostname]
[replica-set-name]/[hostname]:port
```

> **Warning:** Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The `sh.addShard()` (page 1022) method is a helper for the `addShard` (page 874) command. The `addShard` (page 874) command has additional options which are not available with this helper.

Example To add a shard on a replica set, specify the name of the replica set and the hostname of at least one member of the replica set, as a seed. If you specify additional hostnames, all must be members of the same replica set.

The following example adds a replica set named `repl0` and specifies one member of the replica set:

```
sh.addShard("repl0/mongodb3.example.net:27327")
```

sh.addShardTag()

Definition

sh.**addShardTag**(*shard*, *tag*)

New in version 2.2.

Associates a shard with a tag or identifier. MongoDB uses these identifiers to direct *chunks* that fall within a tagged range to specific shards. `sh.addTagRange()` (page 1023) associates chunk ranges with tag ranges.

param string shard The name of the shard to which to give a specific tag.

param string tag The name of the tag to add to the shard.

Always issue `sh.addShardTag()` (page 1023) when connected to a `mongos` (page 1061) instance.

Example The following example adds three tags, `NYC`, `LAX`, and `NRT`, to three shards:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "LAX")
sh.addShardTag("shard0002", "NRT")
```

See also:

`sh.addTagRange()` (page 1023) and `sh.removeShardTag()` (page 1027).

sh.addTagRange()

Definition

sh.**addTagRange**(*namespace*, *minimum*, *maximum*, *tag*)

New in version 2.2.

Attaches a range of shard key values to a shard tag created using the `sh.addShardTag()` (page 1023) method. `sh.addTagRange()` (page 1023) takes the following arguments:

param string namespace The *namespace* of the sharded collection to tag.

param document minimum The minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param document maximum The maximum value of the shard key range to include in the tag. Specify the maximum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.

param string tag The name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

Use `sh.addShardTag()` (page 1023) to ensure that the balancer migrates documents that exist within the specified range to a specific shard or set of shards.

Always issue `sh.addTagRange()` (page 1023) when connected to a `mongos` (page 1061) instance.

Note: If you add a tag range to a collection using `sh.addTagRange()` (page 1023) and then later drop the collection or its database, MongoDB does not remove the tag association. If you later create a new collection with the same name, the old tag association will apply to the new collection.

Example Given a shard key of `{state: 1, zip: 1}`, the following operation creates a tag range covering zip codes in New York State:

```
sh.addTagRange( "exampledb.collection",
                { state: "NY", zip: MinKey },
                { state: "NY", zip: MaxKey },
                "NY"
              )
```

sh.disableBalancing()

Description

`sh.disableBalancing(namespace)`

Disables the balancer for the specified sharded collection. This does not affect the balancing of *chunks* for other sharded collections in the same cluster.

The `sh.disableBalancing()` (page 1024) method has the following parameter:

param string namespace The *namespace* of the collection.

For more information on the balancing process, see *Manage Sharded Cluster Balancer* (page 541) and *Sharded Collection Balancing* (page 510).

sh.enableBalancing()

Description

`sh.enableBalancing(collection)`

Enables the balancer for the specified sharded collection.

The `sh.enableBalancing()` (page 1024) method has the following parameter:

param string collection The *namespace* of the collection.

Important: `sh.enableBalancing()` (page 1024) does not *start* balancing. Rather, it allows balancing of this collection the next time the balancer runs.

For more information on the balancing process, see *Manage Sharded Cluster Balancer* (page 541) and *Sharded Collection Balancing* (page 510).

sh.enableSharding()

Definition

`sh.enableSharding(database)`

Enables sharding on the specified database. This does not automatically shard any collections but makes it possible to begin sharding collections using `sh.shardCollection()` (page 1027).

The `sh.enableSharding()` (page 1024) method has the following parameter:

param string database The name of the database shard. Enclose the name in quotation marks.

See also:

`sh.shardCollection()` (page 1027)

sh.getBalancerHost()

`sh.getBalancerHost()`

Returns String in form *hostname*:*port*

`sh.getBalancerHost()` (page 1025) returns the name of the server that is running the balancer.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

sh.getBalancerState()

`sh.getBalancerState()`

Returns boolean

`sh.getBalancerState()` (page 1025) returns `true` when the *balancer* is enabled and false if the balancer is disabled. This does not reflect the current state of balancing operations: use `sh.isBalancerRunning()` (page 1026) to check the balancer's current state.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

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()` (page 1025) to determine if the balancer is enabled or disabled.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

sh.moveChunk()

Definition

`sh.moveChunk(`*namespace*, *query*, *destination*`)`

Moves the *chunk* that contains the document specified by the `query` to the `destination` shard. `sh.moveChunk()` (page 1026) provides a wrapper around the `moveChunk` (page 880) database command and takes the following arguments:

param string namespace The *namespace* of the sharded collection that contains the chunk to migrate.

param document query An equality match on the shard key that selects the chunk to move.

param string destination The name of the shard to move.

Important: In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` (page 1026) directly.

See also:

`moveChunk` (page 880), `sh.splitAt()` (page 1028), `sh.splitFind()` (page 1028), *Sharding* (page 485), and *chunk migration* (page 511).

Example Given the `people` collection in the `records` database, the following operation finds the chunk that contains the documents with the `zipcode` field set to `53187` and then moves that chunk to the shard named `shard0019`:

```
sh.moveChunk("records.people", { zipcode: 53187 }, "shard0019")
```

sh.removeShardTag()

Definition

`sh.removeShardTag(shard, tag)`

New in version 2.2.

Removes the association between a tag and a shard. Always issue `sh.removeShardTag()` (page 1027) when connected to a `mongos` (page 1061) instance.

param string shard The name of the shard from which to remove a tag.

param string tag The name of the tag to remove from the shard.

See also:

`sh.addShardTag()` (page 1023), `sh.addTagRange()` (page 1023)

sh.setBalancerState()

Description

`sh.setBalancerState(state)`

Enables or disables the *balancer*. Use `sh.getBalancerState()` (page 1025) to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` (page 1026) to check its current state.

The `sh.getBalancerState()` (page 1025) method has the following parameter:

param Boolean state Set this to `true` to enable the balancer and `false` to disable it.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

sh.shardCollection()

Definition

`sh.shardCollection(namespace, key, unique)`

Shards a collection using the `key` as a the *shard key*. `sh.shardCollection()` (page 1027) takes the following arguments:

param string namespace The *namespace* of the collection to shard.

param document key A *document* that specifies the *shard key* to use to *partition* and distribute objects among the shards. A shard key may be one field or multiple fields. A shard key with multiple fields is called a "compound shard key."

param Boolean unique When true, ensures that the underlying index enforces a unique constraint. *Hashed shard keys* do not support unique constraints.

New in version 2.4: Use the form `{field: "hashed"}` to create a *hashed shard key*. Hashed shard keys may not be compound indexes.

> **Warning:** MongoDB provides no method to deactivate sharding for a collection after calling `shardCollection` (page 876). Additionally, after `shardCollection` (page 876), you cannot change shard keys or modify the value of any field used in your shard key index.

See also:

`shardCollection` (page 876) for additional options, *Sharding* (page 485) and *Sharding Introduction* (page 487) for an overview of sharding, *Deploy a Sharded Cluster* (page 516) for a tutorial, and *Shard Keys* (page 502) for choosing a shard key.

Example Given the `people` collection in the `records` database, the following command shards the collection by the `zipcode` field:

```
sh.shardCollection("records.people", { zipcode: 1} )
```

sh.splitAt()

Definition

`sh.splitAt(namespace, query)`

Splits the chunk containing the document specified by the query as if that document were at the "middle" of the collection, even if the specified document is not the actual median of the collection.

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Use this command to manually split chunks unevenly. Use the "`sh.splitFind()` (page 1028)" 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()` (page 1028).

sh.splitFind()

Definition

`sh.splitFind(namespace, query)`

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` (page 1028) 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()` (page 1028).

param string namespace The namespace (i.e. `<database>.<collection>`) of the sharded collection that contains the chunk to split.

param document query A query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

sh.startBalancer()

Definition

`sh.startBalancer(timeout, interval)`

Enables the balancer in a sharded cluster and waits for balancing to initiate.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep each cycle of waiting.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.setBalancerState()` (page 1027)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

sh.status()

Definition

`sh.status()`

Prints a formatted report of the sharding configuration and the information regarding existing chunks in a *sharded cluster*. The default behavior suppresses the detailed chunk information if the total number of chunks is greater than or equal to 20.

The `sh.status()` (page 1029) method has the following parameter:

param Boolean verbose If `true`, the method displays details of the document distribution across chunks when you have 20 or more chunks.

See also:

`db.printShardingStatus()` (page 1010)

Output Examples The *Sharding Version* (page 1030) section displays information on the *config database*:

```
--- Sharding Status ---
  sharding version: {
   "_id" : <num>,
   "version" : <num>,
   "minCompatibleVersion" : <num>,
   "currentVersion" : <num>,
   "clusterId" : <ObjectId>
}
```

The *Shards* (page 1031) section lists information on the shard(s). For each shard, the section displays the name, host, and the associated tags, if any.

```
shards:
 {  "_id" : <shard name1>,
    "host" : <string>,
    "tags" : [ <string> ... ]
 }
 {  "_id" : <shard name2>,
    "host" : <string>,
    "tags" : [ <string> ... ]
 }
 ...
```

The *Databases* (page 1031) section lists information on the database(s). For each database, the section displays the name, whether the database has sharding enabled, and the *primary shard* for the database.

```
databases:
 {  "_id" : <dbname1>,
    "partitioned" : <boolean>,
    "primary" : <string>
 }
 {  "_id" : <dbname2>,
    "partitioned" : <boolean>,
    "primary" : <string>
 }
 ...
```

The *Sharded Collection* (page 1031) section provides information on the sharding details for sharded collection(s). For each sharded collection, the section displays the shard key, the number of chunks per shard(s), the distribution of documents across chunks [3], and the tag information, if any, for shard key range(s).

```
<dbname>.<collection>
   shard key: { <shard key> : <1 or hashed> }
   chunks:
      <shard name1> <number of chunks>
      <shard name2> <number of chunks>
      ...
   { <shard key>: <min range1> } -->> { <shard key> : <max range1> } on : <shard name> <last modified
   { <shard key>: <min range2> } -->> { <shard key> : <max range2> } on : <shard name> <last modified
   ...
   tag: <tag1>  { <shard key> : <min range1> } -->> { <shard key> : <max range1> }
   ...
```

Output Fields

Sharding Version

sh.status.sharding-version._id

The `_id` (page 1030) is an identifier for the version details.

sh.status.sharding-version.version

The `version` (page 1030) is the version of the *config server* for the sharded cluster.

sh.status.sharding-version.minCompatibleVersion

The `minCompatibleVersion` (page 1030) is the minimum compatible version of the config server.

[3] The sharded collection section, by default, displays the chunk information if the total number of chunks is less than 20. To display the information when you have 20 or more chunks, call the `sh.status()` (page 1029) methods with the `verbose` parameter set to `true`, i.e. `sh.status(true)`.

`sh.status.sharding-version.currentVersion`
The `currentVersion` (page 1030) is the current version of the config server.

`sh.status.sharding-version.clusterId`
The `clusterId` (page 1031) is the identification for the sharded cluster.

Shards

`sh.status.shards._id`
The `_id` (page 1031) displays the name of the shard.

`sh.status.shards.host`
The `host` (page 1031) displays the host location of the shard.

`sh.status.shards.tags`
The `tags` (page 1031) displays all the tags for the shard. The field only displays if the shard has tags.

Databases

`sh.status.databases._id`
The `_id` (page 1031) displays the name of the database.

`sh.status.databases.partitioned`
The `partitioned` (page 1031) displays whether the database has sharding enabled. If `true`, the database has sharding enabled.

`sh.status.databases.primary`
The `primary` (page 1031) displays the *primary shard* for the database.

Sharded Collection

`sh.status.databases.shard-key`
The `shard-key` (page 1031) displays the shard key specification document.

`sh.status.databases.chunks`
The `chunks` (page 1031) lists all the shards and the number of chunks that reside on each shard.

`sh.status.databases.chunk-details`
The `chunk-details` (page 1031) lists the details of the chunks [1]:

- The range of shard key values that define the chunk,
- The shard where the chunk resides, and
- The last modified timestamp for the chunk.

`sh.status.databases.tag`
The `tag` (page 1031) lists the details of the tags associated with a range of shard key values.

sh.stopBalancer()

Definition

`sh.stopBalancer(timeout, interval)`
Disables the balancer in a sharded cluster and waits for balancing to complete.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep each cycle of waiting.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)

- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.waitForBalancer()` (page 1032)
- `sh.waitForBalancerOff()` (page 1032)

sh.waitForBalancer()

Definition

`sh.waitForBalancer(wait, timeout, interval)`

Waits for a change in the state of the balancer. `sh.waitForBalancer()` (page 1032) is an internal method, which takes the following arguments:

param Boolean wait Set to `true` to ensure the balancer is now active. The default is `false`, which waits until balancing stops and becomes inactive.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep.

sh.waitForBalancerOff()

Definition

`sh.waitForBalancerOff(timeout, interval)`

Internal method that waits until the balancer is not running.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep.

See also:

- `sh.enableBalancing()` (page 1024)
- `sh.disableBalancing()` (page 1024)
- `sh.getBalancerHost()` (page 1025)
- `sh.getBalancerState()` (page 1025)
- `sh.isBalancerRunning()` (page 1026)
- `sh.setBalancerState()` (page 1027)
- `sh.startBalancer()` (page 1029)
- `sh.stopBalancer()` (page 1031)
- `sh.waitForBalancer()` (page 1032)

sh.waitForDLock()

Definition

`sh.waitForDLock(lockname, wait, timeout, interval)`

Waits until the specified distributed lock changes state. `sh.waitForDLock()` (page 1033) is an internal method that takes the following arguments:

param string lockname The name of the distributed lock.

param Boolean wait Set to `true` to ensure the balancer is now active. Set to `false` to wait until balancing stops and becomes inactive.

param integer timeout Milliseconds to wait.

param integer interval Milliseconds to sleep in each waiting cycle.

sh.waitForPingChange()

Definition

`sh.waitForPingChange(activePings, timeout, interval)`

`sh.waitForPingChange()` (page 1033) waits for a change in ping state of one of the `activepings`, and only returns when the specified ping changes state.

param array activePings An array of active pings from the `mongos` (page 559) collection.

param integer timeout Number of milliseconds to wait for a change in ping state.

param integer interval Number of milliseconds to sleep in each waiting cycle.

55.3.6 Connection

Connection Methods

Name	Description
`Mongo.getDB()` (page 1033)	Returns a database object.
`Mongo.getReadPrefMode()` (page 1034)	Returns the current read preference mode for the MongoDB connection.
`Mongo.getReadPrefTagSet()` (page 1034)	Returns the read preference tag set for the MongoDB connection.
`Mongo.setReadPref()` (page 1035)	Sets the *read preference* for the MongoDB connection.
`Mongo.setSlaveOk()` (page 1035)	Allows operations on the current connection to read from *secondary* members.
`Mongo()` (page 1036)	Creates a new connection object.
`connect()`	Connects to a MongoDB instance and to a specified database on that instance.

Mongo.getDB()

Description

`Mongo.getDB(<database>)`

Provides access to database objects from the `mongo` (page 1066) shell or from a JavaScript file.

The `Mongo.getDB()` (page 1033) method has the following parameter:

param string database The name of the database to access.

Example The following example instantiates a new connection to the MongoDB instance running on the localhost interface and returns a reference to "myDatabase":

```
db = new Mongo().getDB("myDatabase");
```

See also:

`Mongo()` (page 1036) and *connect()* (page 1036)

Mongo.getReadPrefMode()

`Mongo.getReadPrefMode()`

Returns The current *read preference* mode for the `Mongo()` (page 1005) connection object.

See *Read Preference* (page 398) for an introduction to read preferences in MongoDB. Use `getReadPrefMode()` (page 1034) to return the current read preference mode, as in the following example:

```
db.getMongo().getReadPrefMode()
```

Use the following operation to return and print the current read preference mode:

```
print(db.getMongo().getReadPrefMode());
```

This operation will return one of the following read preference modes:

- `primary` (page 400)
- `primaryPreferred` (page 400)
- `secondary` (page 400)
- `secondaryPreferred` (page 400)
- `nearest` (page 400)

See also:

Read Preference (page 398), `cursor.readPref()` (page 989), `Mongo.setReadPref()` (page 1035), and `Mongo.getReadPrefTagSet()` (page 1034).

Mongo.getReadPrefTagSet()

`Mongo.getReadPrefTagSet()`

Returns The current *read preference* tag set for the `Mongo()` (page 1005) connection object.

See *Read Preference* (page 398) for an introduction to read preferences and tag sets in MongoDB. Use `getReadPrefTagSet()` (page 1034) to return the current read preference tag set, as in the following example:

```
db.getMongo().getReadPrefTagSet()
```

Use the following operation to return and print the current read preference tag set:

```
printjson(db.getMongo().getReadPrefTagSet());
```

See also:

Read Preference (page 398), `cursor.readPref()` (page 989), `Mongo.setReadPref()` (page 1035), and `Mongo.getReadPrefTagSet()` (page 1034).

Mongo.setReadPref()

Definition

`Mongo.setReadPref(mode, tagSet)`

Call the `setReadPref()` (page 1035) method on a `Mongo` (page 1005) connection object to control how the client will route all queries to members of the replica set.

param string mode One of the following *read preference* modes: `primary` (page 400), `primaryPreferred` (page 400), `secondary` (page 400), `secondaryPreferred` (page 400), or `nearest` (page 400).

param array tagSet A tag set used to specify custom read preference modes. For details, see *Tag Sets* (page 401).

Examples To set a read preference mode in the `mongo` (page 1066) shell, use the following operation:

```
db.getMongo().setReadPref('primaryPreferred')
```

To set a read preference that uses a tag set, specify an array of tag sets as the second argument to `Mongo.setReadPref()` (page 1035), as in the following:

```
db.getMongo().setReadPref('primaryPreferred', [ { "dc": "east" } ] )
```

You can specify multiple tag sets, in order of preference, as in the following:

```
db.getMongo().setReadPref('secondaryPreferred',
                          [ { "dc": "east", "use": "production" },
                            { "dc": "east", "use": "reporting" },
                            { "dc": "east" },
                            {}
                          ] )
```

If the replica set cannot satisfy the first tag set, the client will attempt to use the second read preference. Each tag set can contain zero or more field/value tag pairs, with an "empty" document acting as a wildcard which matches a replica set member with any tag set or no tag set.

Note: You must call `Mongo.setReadPref()` (page 1035) on the connection object before retrieving documents using that connection to use that read preference.

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()` (page 1017).

See the `readPref()` (page 989) method for more fine-grained control over *read preference* (page 398) in the `mongo` (page 1066) shell.

Mongo()

Description

Mongo (*host*)

JavaScript constructor to instantiate a database connection from the `mongo` (page 1066) shell or from a JavaScript file.

The `Mongo()` (page 1036) method has the following parameter:

param string host The host, either in the form of `<host>` or `<host><:port>`.

Instantiation Options Use the constructor without a parameter to instantiate a connection to the localhost interface on the default port.

Pass the `<host>` parameter to the constructor to instantiate a connection to the `<host>` and the default port.

Pass the `<host><:port>` parameter to the constructor to instantiate a connection to the `<host>` and the `<port>`.

See also:

`Mongo.getDB()` (page 1033) and `db.getMongo()` (page 1005).

connect()

connect (*<hostname><:port>/<database>*)

The `connect()` method creates a connection to a MongoDB instance. However, use the `Mongo()` (page 1036) object and its `getDB()` (page 1033) method in most cases.

`connect()` accepts a string `<hostname><:port>/<database>` parameter to connect to the MongoDB instance on the `<hostname><:port>` and return the reference to the database `<database>`.

The following example instantiates a new connection to the MongoDB instance running on the localhost interface and returns a reference to `myDatabase`:

```
db = connect("localhost:27017/myDatabase")
```

See also:

`Mongo.getDB()` (page 1033)

55.3.7 Subprocess

Subprocess Methods

Name	Description
`clearRawMongoProgramOutput()` (page 1037)	For internal use.
`rawMongoProgramOutput()` (page 1037)	For internal use.
`run()`	For internal use.
`runMongoProgram()` (page 1037)	For internal use.
`runProgram()` (page 1037)	For internal use.
`startMongoProgram()`	For internal use.
`stopMongoProgram()` (page 1037)	For internal use.
`stopMongoProgramByPid()` (page 1037)	For internal use.
`stopMongod()` (page 1037)	For internal use.
`waitMongoProgramOnPort()` (page 1038)	For internal use.
`waitProgram()` (page 1038)	For internal use.

clearRawMongoProgramOutput()

clearRawMongoProgramOutput()
For internal use.

rawMongoProgramOutput()

rawMongoProgramOutput()
For internal use.

run()

run()
For internal use.

runMongoProgram()

runMongoProgram()
For internal use.

runProgram()

runProgram()
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.

55.3.8 Native

Native Methods

Name	Description
cat()	Returns the contents of the specified file.
cd()	Changes the current working directory to the specified path.
copyDbpath() (page 1039)	Copies a local dbpath (page 1118). For internal use.
resetDbpath() (page 1039)	Removes a local dbpath (page 1118). For internal use.
fuzzFile() (page 1039)	For internal use to support testing.
getHostName() (page 1039)	Returns the hostname of the system running the mongo (page 1066) shell.
getMemInfo() (page 1039)	Returns a document that reports the amount of memory used by the shell.
hostname()	Returns the hostname of the system running the shell.
_isWindows() (page 1040)	Returns true if the shell runs on a Windows system; false if a Unix or Linux system.
listFiles() (page 1040)	Returns an array of documents that give the name and size of each object in the directory.
load()	Loads and runs a JavaScript file in the shell.
ls()	Returns a list of the files in the current directory.
md5sumFile() (page 1040)	The *md5* hash of the specified file.
mkdir()	Creates a directory at the specified path.
pwd()	Returns the current directory.
quit()	Exits the current shell session.
_rand() (page 1041)	Returns a random number between 0 and 1.
removeFile() (page 1041)	Removes the specified file from the local file system.
_srand() (page 1042)	For internal use.

cat()

Definition

cat(*filename*)
Returns the contents of the specified file. The method returns with output relative to the current shell session and does not impact the server.

param string filename Specify a path and file name on the local file system.

cd()

Definition

cd(*path*)

param string path A path on the file system local to the `mongo` (page 1066) shell context.

`cd()` changes the directory context of the `mongo` (page 1066) shell and has no effect on the MongoDB server.

copyDbpath()

copyDbpath()

For internal use.

resetDbpath()

resetDbpath()

For internal use.

fuzzFile()

Description

fuzzFile(*filename*)

For internal use.

param string filename A filename or path to a local file.

getHostName()

getHostName()

Returns The hostname of the system running the `mongo` (page 1066) 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*.

hostname()

hostname()

Returns The hostname of the system running the `mongo` (page 1066) shell process.

_isWindows()

_isWindows()

Returns boolean.

Returns "true" if the mongo (page 1066) shell is running on a system that is Windows, or "false" if the shell 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 (page 1066) 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()

Description

load(*file*)

Loads and runs a JavaScript file into the current shell environment.

The `load()` method has the following parameter:

param string file Specify a path and file name containing JavaScript.

To run JavaScript with the mongo (page 1066) shell, you can either:

- use the `--eval` option when invoking the shell to evaluate a small amount of JavaScript code, or
- specify a file name with *mongo* (page 1069). mongo (page 1066) will execute the script and then exit. Add the `--shell` 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 (page 1066) 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()

Description

md5sumFile (*filename*)

Returns a *md5* hash of the specified file.

The `md5sumFile()` (page 1040) method has the following parameter:

param string filename A file name.

Note: The specified filename must refer to a file located on the system running the `mongo` (page 1066) shell.

mkdir()

Description

mkdir (*path*)

Creates a directory at the specified path. This method creates the entire path specified if the enclosing directory or directories do not already exit.

This method is equivalent to **mkdir -p** with BSD or GNU utilities.

The `mkdir()` method has the following parameter:

param string path A path on the local filesystem.

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.

removeFile()

Description

removeFile (*filename*)

Removes the specified file from the local file system.

The `removeFile()` (page 1041) method has the following parameter:

param string filename A filename or path to a local file.

_srand()

_srand()
For internal use.

55.4 SQL to MongoDB Mapping Chart

In addition to the charts that follow, you might want to consider the *Frequently Asked Questions* (page 731) section for a selection of common questions about MongoDB.

55.4.1 Executables

The following table presents the MySQL/Oracle executables and the corresponding MongoDB executables.

	MySQL/Oracle	MongoDB
Database Server	`mysqld/oracle`	`mongod` (page 1049)
Database Client	`mysql/sqlplus`	`mongo` (page 1066)

55.4.2 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 287).

55.4.3 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` (page 961) 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()` (page 961) and `createCollection()` (page 997) for more information.

`ALTER TABLE users`
`ADD join_date DATETIME`
	Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` (page 974) operations can add fields to existing documents using the `$set` (page 814) operator.
	`db.users.update(`
	`    { },`
	`    { $set: { join_date: new Date() } },`
	`    { multi: true }`
	`)`
		See the *Data Modeling Considerations for MongoDB Applications* (page 107), `update()` (page 974), and `$set` (page 814) 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 its documents; i.e. there is no structural alteration at the collection level.
	However, at the document level, `update()` (page 974) operations can remove fields from documents using the `$unset` (page 814) operator.
	`db.users.update(`
	`    { },`
	`    { $unset: { join_date: "" } },`
	`    { multi: true }`
	`)`
		See *Data Modeling Considerations for MongoDB Applications* (page 107), `update()` (page 974), and `$unset` (page 814) 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()` (page 949) and *indexes* (page 309) 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()` (page 949) and *indexes* (page 309) for more information.

`DROP TABLE users`	`db.users.drop()`	See `drop()` (page 948) for more information.

Insert

The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.

SQL INSERT Statements	MongoDB insert() Statements	Reference

<pre>INSERT INTO users(user_id,
                  age,
                  status)
VALUES ("bcd001",
        45,
        "A")</pre>
	<pre>db.users.insert({
	       user_id: "bcd001",
	       age: 45,
	       status: "A"
	})</pre>
		See `insert()` (page 961) 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() (page 951) for more information.

SELECT id, user_id, status
FROM users
	db.users.find(
	    { },
	    { user_id: 1, status: 1 }
	)
		See find() (page 951) for more information.

SELECT user_id, status
FROM users
	db.users.find(
	    { },
	    { user_id: 1, status: 1, _id: 0 }
	)
		See find() (page 951) for more information.

SELECT * FROM users WHERE status = "A"	db.users.find({ status: "A" })	See find() (page 951) 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() (page 951) for more information.

SELECT * FROM users WHERE status != "A"	db.users.find({ status: { $ne: "A" } })	See find() (page 951) and $ne (page 789) for more information.

SELECT *
FROM users
WHERE status = "A"
AND age = 50
	db.users.find(
	    { status: "A",
	      age: 50 }
	)
		See find() (page 951) and $and (page 791) for more information.

SELECT *
FROM users
WHERE status = "A"
OR age = 50
	db.users.find(
	    { $or: [{ status: "A" } ,
	            { age: 50 }] }
	)
		See find() (page 951) and $or (page 790) for more information.

SELECT * FROM users WHERE age > 25	db.users.find({ age: { $gt: 25 } })	See find() (page 951) and $gt (page 786) for more information.
SELECT * FROM users WHERE age < 25	db.users.find({ age: { $lt: 25 } })	See find() (page 951) and $lt (page 788) for more information.
SELECT * FROM users WHERE age > 25 AND age <= 50	db.users.find({ age: { $gt: 25, $lte: 50 } })	See find() (page 951), $gt (page 786), and $lte (page 788) for more information.
SELECT * FROM users WHERE user_id like "%bc%"	db.users.find({ user_id: /bc/ })	See find() (page 951) and $regex (page 798) 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 } }, { $set: { status: "C" } }, { multi: true } )`	See `update()` (page 974), `$gt` (page 786), and `$set` (page 814) for more information.
`UPDATE users SET age = age + 3 WHERE status = "A"`	`db.users.update( { status: "A" } , { $inc: { age: 3 } }, { multi: true } )`	See `update()` (page 974), `$inc` (page 810), and `$set` (page 814) 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()` (page 970) for more information.
`DELETE FROM users`	`db.users.remove( )`	See `remove()` (page 970) for more information.

CHAPTER 56

Architecture and Components

56.1 MongoDB Package Components

56.1.1 Core Processes

The core components in the MongoDB package are: `mongod` (page 1049), the core database process; `mongos` (page 1061) the controller and query router for *sharded clusters*; and `mongo` (page 1066) the interactive MongoDB Shell.

mongod

Synopsis

`mongod` (page 1049) 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` (page 1049). These options are primarily useful for testing purposes. In common operation, use the *configuration file options* (page 1115) to control the behavior of your database, which is fully capable of all operations described below.

Options

mongod

Core Options
mongod
command line option!–help, -h

--help, -h
Returns a basic help and usage text.

command line option!–version

--version
Returns the version of the `mongod` (page 1049) daemon.

command line option!–config <filename>, -f <filename>

--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 1115)" document for more information about these options.

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in the log file specified by *--logpath* (page 1063). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

command line option!–quiet

--quiet
Runs the `mongod` (page 1049) instance in a quiet mode that attempts to limit the amount of output. This option suppresses:

- output from *database commands*, including `drop` (page 884), `dropIndexes` (page 888), `diagLogging` (page 906), `validate` (page 908), and `clean` (page 890).
- replication activity.
- connection accepted events.
- connection closed events.

command line option!–port <port>

--port <port>
Specifies a TCP port for the `mongod` (page 1049) to listen for client connections. By default `mongod` (page 1049) listens for connections on port 27017.

UNIX-like systems require root privileges to use ports with numbers lower than 1024.

command line option!–bind_ip <ip address>

--bind_ip <ip address>
The IP address that the `mongod` (page 1049) process will bind to and listen for connections. By default `mongod` (page 1049) listens for connections all interfaces. You may attach `mongod` (page 1049) to any interface; however, when attaching `mongod` (page 1049) to a publicly accessible interface ensure that you have implemented proper authentication and/or firewall restrictions to protect the integrity of your database.

command line option!–maxConns <number>

--maxConns <number>
Specifies the maximum number of simultaneous connections that `mongod` (page 1049) 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 1116) to a value higher than *20000*.

command line option!–objcheck

--objcheck
Forces the `mongod` (page 1049) to validate all requests from clients upon receipt to ensure that clients never insert invalid documents into the database. For objects with a high degree of sub-document nesting, *--objcheck* (page 1107) can have a small impact on performance. You can set *--noobjcheck* (page 1082) to disable object checking at run-time.

Changed in version 2.4: MongoDB enables *--objcheck* (page 1107) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–noobjcheck

--noobjcheck
New in version 2.4.

Disables the default document validation that MongoDB performs on all incoming BSON documents.

command line option!–logpath <path>

--logpath `<path>`
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, `mongod` (page 1049) will output all log information to the standard output. Additionally, unless you also specify *--logappend* (page 1063), 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.

command line option!–logappend

--logappend
When specified, this option ensures that `mongod` (page 1049) appends new entries to the end of the logfile rather than overwriting the content of the log when the process restarts.

command line option!–syslog

--syslog
New in version 2.1.0.

Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with *--logpath* (page 1063).

Warning: You cannot use *--syslog* (page 1063) with *--logpath* (page 1063).

command line option!–pidfilepath <path>

--pidfilepath `<path>`
Specify a file location to hold the "*PID*" or process ID of the `mongod` (page 1049) process. Useful for tracking the `mongod` (page 1049) process in combination with the *mongod --fork* option.

Without a specified *--pidfilepath* (page 1063) option, `mongos` (page 1061) creates no PID file.

command line option!–keyFile <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 212)" and "*Replica Set Tutorials* (page 415)."

command line option!–nounixsocket

--nounixsocket
Disables listening on the UNIX socket. `mongod` (page 1049) always listens on the UNIX socket, unless *--nounixsocket* (page 1064) is set, *--bind_ip* (page 1062) is *not* set, or *--bind_ip* (page 1062) does *not* specify `127.0.0.1`.

command line option!–unixSocketPrefix <path>

--unixSocketPrefix <path>
Specifies a path for the UNIX socket. Unless this option has a value `mongod` (page 1049) creates a socket with `http://docs.mongodb.org/manualtmp` as a prefix.

MongoDB will *always* create and listen on a UNIX socket, unless *--nounixsocket* (page 1064) is set, *--bind_ip* (page 1062) is *not* set, or *--bind_ip* (page 1062) does *not* specify `127.0.0.1`.

command line option!–fork

--fork
Enables a *daemon* mode for `mongod` (page 1049) 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.

command line option!–auth

--auth
Enables database authentication for users connecting from remote hosts. Configure users via the *mongo shell* (page 1066). 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* (page 207) page for more information regarding this functionality.

command line option!–cpu

--cpu
Forces `mongod` (page 1049) to report the percentage of CPU time in write lock. `mongod` (page 1049) generates output every four seconds. MongoDB writes this data to standard output or the logfile if using the `logpath` (page 1116) option.

command line option!–dbpath <path>

--dbpath <path>
Specify a directory for the `mongod` (page 1049) instance to store its data. Typical locations include: `http://docs.mongodb.org/manualsrv/mongodb`, `http://docs.mongodb.org/manualvar/lib/mongodb` or `http://docs.mongodb.org/manualopt/mongo`

Unless specified, `mongod` (page 1049) will look for data files in the default `http://docs.mongodb.org/manualdata/db` directory. (Windows systems use the `\data\db` directory.) If you installed using a package management system. Check the `/etc/mongodb.conf` file provided by your packages to see the configuration of the `dbpath` (page 1118).

command line option!–diaglog <value>

--diaglog <value>
Creates a very verbose, *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the `dbpath` (page 1118) 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` (page 1106) tool to replay this output for investigation. Given a typical diaglog file, located at `http://docs.mongodb.org/manualdata/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 1052) is for internal use and not intended for most users.

> **Warning:** Setting the diagnostic level to `0` will cause `mongod` (page 1049) to stop writing data to the *diagnostic log* file. However, the `mongod` (page 1049) 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` (page 1049) instance before doing so.

command line option!–directoryperdb

--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 1081) 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.

> **Warning:** If you have an existing `mongod` (page 1049) instance and `dbpath` (page 1118), and you want to enable *--directoryperdb* (page 1081), you **must** migrate your existing databases to directories before setting *--directoryperdb* (page 1081) to access those databases.

Example

Given a `dbpath` (page 1118) directory with the following items:

```
journal
mongod.lock
local.0
local.1
local.ns
test.0
test.1
test.ns
```

To enable *--directoryperdb* (page 1081) you would need to modify the `dbpath` (page 1118) to resemble the following:

```
journal
mongod.lock
local/local.0
local/local.1
local/local.ns
test/test.0
test/test.1
test/test.ns
```

command line option!–journal

--journal

Enables operation journaling to ensure write durability and data consistency. `mongod` (page 1049) enables journaling by default on 64-bit builds of versions after 2.0.

command line option!–journalOptions <arguments>

--journalOptions <arguments>

Provides functionality for testing. Not for general use, and may affect database integrity.

command line option!–journalCommitInterval <value>

--journalCommitInterval <value>
Specifies the maximum amount of time for `mongod` (page 1049) to allow between journal operations. Possible values are between 2 and 300 milliseconds. Lower values increase the durability of the journal, at the expense of disk performance.

The default journal commit interval is 100 milliseconds if a single block device (e.g. physical volume, RAID device, or LVM volume) contains both the journal and the data files.

If different block devices provide the journal and data files the default journal commit interval is 30 milliseconds.

To force `mongod` (page 1049) to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` is pending, `mongod` (page 1049) will reduce `journalCommitInterval` (page 1119) to a third of the set value.

command line option!–ipv6

--ipv6
Specify this option to enable IPv6 support. This will allow clients to connect to `mongod` (page 1049) using IPv6 networks. `mongod` (page 1049) disables IPv6 support by default in `mongod` (page 1049) and all utilities.

command line option!–jsonp

--jsonp
Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

command line option!–noauth

--noauth
Disable authentication. Currently the default. Exists for future compatibility and clarity.

command line option!–nohttpinterface

--nohttpinterface
Disables the HTTP interface.

command line option!–nojournal

--nojournal
Disables the durability journaling. By default, `mongod` (page 1049) enables journaling in 64-bit versions after v2.0.

command line option!–noprealloc

--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.

command line option!–noscripting

--noscripting
Disables the scripting engine.

command line option!–notablescan

--notablescan
Forbids operations that require a table scan.

command line option!–nssize <value>

--nssize <value>
Specifies the default size for namespace files (i.e `.ns`). This option has no impact on the size of existing namespace files. The maximum size is 2047 megabytes.

The default value is 16 megabytes; this provides for approximately 24,000 namespaces. Each collection, as well as each index, counts as a namespace.

command line option!–profile <level>

--profile <level>

Changes the level of database profiling, which inserts information about operation performance into output of mongod (page 1049) 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.

command line option!–quota

--quota

Enables a maximum limit for the number data files each database can have. When running with *--quota* (page 1055), there are a maximum of 8 data files per database. Adjust the quota with the *--quotaFiles* (page 1055) option.

command line option!–quotaFiles <number>

--quotaFiles <number>

Modify limit on the number of data files per database. This option requires the *--quota* (page 1055) setting. The default value for *--quotaFiles* (page 1055) is 8.

command line option!–rest

--rest

Enables the simple *REST* API.

command line option!–repair

--repair

Runs a repair routine on all databases. This is equivalent to shutting down and running the repairDatabase (page 895) 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 (page 895) or related options like db.repairDatabase() (page 1011) in the mongo (page 1066) shell or *mongod --repair*. Restore from an intact copy of your data.

Note: When using *journaling*, there is almost never any need to run repairDatabase (page 895). 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 (page 1049) will refuse to start. In these cases you should start mongod (page 1049) without the *--repair* (page 1078) option to allow mongod (page 1049) 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 (page 1049) cleanly and restart with the *--repair* (page 1078) option.

Note: *--repair* (page 1078) copies data from the source data files into new data files in the repairpath (page 1121), and then replaces the original data files with the repaired data files. *If* `repairpath` (page 1121) is on the same device as `dbpath` (page 1118), you *may* interrupt a `mongod` (page 1049) running *`--repair`* (page 1078) without affecting the integrity of the data set.

---

command line option!–repairpath <path>

`--repairpath` `<path>`
Specifies the root directory containing MongoDB data files, to use for the *`--repair`* (page 1078) operation. Defaults to a `_tmp` directory within the `dbpath` (page 1118).

command line option!–setParameter <options>

`--setParameter` `<options>`
New in version 2.4.

Specifies an option to configure on startup. Specify multiple options with multiple *`--setParameter`* (page 1063) options. See *mongod Parameters* (page 1129) for full documentation of these parameters. The `setParameter` (page 894) database command provides access to many of these parameters. *`--setParameter`* (page 1063) supports the following options:

- `enableLocalhostAuthBypass` (page 1129)
- `enableTestCommands` (page 1129)
- `journalCommitInterval` (page 1129)
- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `replApplyBatchSize` (page 1130)
- `replIndexPrefetch` (page 1130)
- `supportCompatibilityFormPrivilegeDocuments` (page 1131)
- `syncdelay` (page 1131)
- `textSearchEnabled` (page 1132)
- `traceExceptions` (page 1131)

command line option!–slowms <value>

`--slowms` `<value>`
Defines the value of "slow," for the *`--profile`* (page 1055) 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` (page 1049) the profiler writes to the `system.profile` collection. See the `profile` (page 907) command for more information on the database profiler.

command line option!–smallfiles

`--smallfiles`
Enables a mode where MongoDB uses a smaller default file size. Specifically, *`--smallfiles`* (page 1056) reduces the initial size for data files and limits them to 512 megabytes. *`--smallfiles`* (page 1056) also reduces the size of each *journal* files from 1 gigabyte to 128 megabytes.

Use *`--smallfiles`* (page 1056) if you have a large number of databases that each holds a small quantity of data. *`--smallfiles`* (page 1056) can lead your `mongod` (page 1049) to create a large number of files, which may affect performance for larger databases.

command line option!–shutdown

--shutdown

Used in *control scripts*, the *--shutdown* (page 1057) will cleanly and safely terminate the mongod (page 1049) process. When invoking mongod (page 1049) with this option you must set the *--dbpath* (page 1081) option either directly or by way of the *configuration file* (page 1115) and the *--config* (page 1062) option.

--shutdown (page 1057) is only available on Linux systems.

command line option!–syncdelay <value>

--syncdelay <value>

mongod (page 1049) writes data very quickly to the journal, and lazily to the data files. *--syncdelay* (page 1057) controls how much time can pass before MongoDB flushes data to the *database files* via an *fsync* operation. The default setting is 60 seconds. In almost every situation you should not set this value and use the default setting.

The serverStatus (page 919) command reports the background flush thread's status via the backgroundFlushing (page 926) field.

syncdelay (page 1122) has no effect on the journal (page 1119) files or *journaling* (page 155).

> **Warning:** If you set *--syncdelay* (page 1057) to 0, MongoDB will not sync the memory mapped files to disk. Do not set this value on production systems.

command line option!–sysinfo

--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.

command line option!–upgrade

--upgrade

Upgrades the on-disk data format of the files specified by the *--dbpath* (page 1081) to the latest version, if needed.

This option only affects the operation of mongod (page 1049) 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.

command line option!–traceExceptions

--traceExceptions

For internal diagnostic use only.

Replication Options command line option!–replSet <setname>

--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 367)," "*Replica Set Tutorials* (page 415)," and "*Replica Set Configuration* (page 473)"

command line option!–oplogSize <value>

--oplogSize <value>

Specifies a maximum size in megabytes for the replication operation log (e.g. *oplog*.) By mongod (page 1049) 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 (page 1049) has created the oplog for the first time, changing *--oplogSize* (page 1057) will not affect the size of the oplog.

command line option!–fastsync

--fastsync

In the context of *replica set* replication, set this option if you have seeded this member with a snapshot of the *dbpath* of another member of the set. Otherwise the mongod (page 1049) will attempt to perform an initial sync, as though the member were a new member.

> **Warning:** If the data is not perfectly synchronized *and* mongod (page 1049) starts with fastsync (page 1124), then the secondary or slave will be permanently out of sync with the primary, which may cause significant consistency problems.

command line option!–replIndexPrefetch

--replIndexPrefetch

New in version 2.2.

You must use *--replIndexPrefetch* (page 1058) in conjunction with replSet (page 1124). 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 (page 1049) 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.
command line option!–master

--master

Configures mongod (page 1049) to run as a replication *master*.

command line option!–slave

--slave

Configures mongod (page 1049) to run as a replication *slave*.

command line option!–source <host><:port>

--source <host><:port>

For use with the *--slave* (page 1058) option, the `--source` option designates the server that this instance will replicate.

command line option!–only <arg>

--only <arg>

For use with the *--slave* (page 1058) option, the `--only` option specifies only a single *database* to replicate.

command line option!–slavedelay <value>

--slavedelay <value>
For use with the *--slave* (page 1058) option, the `--slavedelay` option configures a "delay" in seconds, for this slave to wait to apply operations from the *master* node.

command line option!–autoresync

--autoresync
For use with the *--slave* (page 1058) option. When set, *--autoresync* (page 1059) option allows this slave to automatically resync if it is more than 10 seconds behind the master. This setting may be problematic if the *--oplogSize* (page 1057) specifies a too small oplog. If the *oplog* is 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 1125) option to `false`, the slave will not attempt an automatic resync more than once in a ten minute period.

Sharding Cluster Options command line option!–configsvr

--configsvr
Declares that this `mongod` (page 1049) 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 a `mongod` (page 1049) with this option is `27019` and the default *--dbpath* (page 1081) directory is `http://docs.mongodb.org/manualdata/configdb`, unless specified.

Changed in version 2.2: *--configsvr* (page 1059) also sets *--smallfiles* (page 1056).

Changed in version 2.4: *--configsvr* (page 1059) creates a local *oplog*.

Do not use *--configsvr* (page 1059) with *--replSet* (page 1057) or *--shardsvr* (page 1059). Config servers cannot be a shard server or part of a *replica set*.

command line option!–shardsvr

--shardsvr
Configures this `mongod` (page 1049) instance as a shard in a partitioned cluster. The default port for these instances is `27018`. The only effect of *--shardsvr* (page 1059) is to change the port number.

SSL Options

See

Connect to MongoDB with SSL (page 179) for full documentation of MongoDB's support.

command line option!–sslOnNormalPorts

--sslOnNormalPorts
New in version 2.2.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Enables SSL for `mongod` (page 1049). With *--sslOnNormalPorts* (page 1059), a `mongod` (page 1049) requires SSL encryption for all connections on the default MongoDB port, or the port specified by *--port* (page 1067). By default, *--sslOnNormalPorts* (page 1059) is disabled.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.2.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains both the SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths

When using `--sslOnNormalPorts` (page 1059), you must specify `--sslPEMKeyFile` (page 1068).

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword `<value>`
New in version 2.2.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the password to de-crypt the certificate-key file (i.e. `--sslPEMKeyFile` (page 1068)). Only use `--sslPEMKeyPassword` (page 1068) if the certificate-key file is encrypted. In all cases, `mongod` (page 1049) will redact the password from all logging and reporting output.

Changed in version 2.4: `--sslPEMKeyPassword` (page 1068) is only needed when the private key is encrypted. In earlier versions `mongod` (page 1049) would require `--sslPEMKeyPassword` (page 1068) whenever using `--sslOnNormalPorts` (page 1059), even when the private key was not encrypted.

command line option!–sslCAFile <filename>

--sslCAFile `<filename>`
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains the root certificate chain from the Certificate Authority. Specify the file name of the `.pem` file using relative or absolute paths

command line option!–sslCRLFile <filename>

--sslCRLFile `<filename>`
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths

command line option!–sslWeakCertificateValidation

--sslWeakCertificateValidation
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Disables the requirement for SSL certificate validation, that `--sslCAFile` (page 1068) enables. With `--sslWeakCertificateValidation` (page 1060), `mongod` (page 1049) will accept connections if the client does not present a certificate when establishing the connection.

If the client presents a certificate and `mongod` (page 1049) has `--sslWeakCertificateValidation` (page 1060) enabled, `mongod` (page 1049) will validate the certificate using the root certificate chain specified by `--sslCAFile` (page 1068), and reject clients with invalid certificates.

Use `--sslWeakCertificateValidation` (page 1060) if you have a mixed deployment that includes clients that do not or cannot present certificates to `mongod` (page 1049).

command line option!–sslFIPSMode

--sslFIPSMode

New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

When specified, `mongod` (page 1049) will use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use `--sslFIPSMode` (page 1061).

Usage

In common usage, the invocation of `mongod` (page 1049) will resemble the following in the context of an initialization or control script:

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

See the "*Configuration File Options* (page 1115)" for more information on how to configure `mongod` (page 1049) using the configuration file.

mongos

Synopsis

`mongos` (page 1061) 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 1061) instance behaves identically to any other MongoDB instance.

Note: Changed in version 2.1.

Some aggregation operations using the `aggregate` (page 834) will cause `mongos` (page 1061) 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.

See also:

Sharding (page 485) and *Sharding Introduction* (page 487).

Options

mongos

mongos

command line option!–help, -h

--help, -h
Returns a basic help and usage text.

command line option!–version

--version
Returns the version of the `mongod` (page 1049) daemon.

command line option!–config <filename>, -f <filename>

--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 1115)" document for more information about these options.

Not all configuration options for `mongod` (page 1049) make sense in the context of `mongos` (page 1061).

command line option!–verbose, -v

--verbose, -v
Increases the amount of internal reporting returned on standard output or in the log file specified by *--logpath* (page 1063). Use the `-v` form to control the level of verbosity by including the option multiple times, (e.g. `-vvvvv`.)

command line option!–quiet

--quiet
Runs the `mongos` (page 1061) instance in a quiet mode that attempts to limit the amount of output.

command line option!–port <port>

--port <port>
Specifies a TCP port for the `mongos` (page 1061) to listen for client connections. By default `mongos` (page 1061) listens for connections on port 27017.

UNIX-like systems require root access to access ports with numbers lower than 1024.

command line option!–bind_ip <ip address>

--bind_ip <ip address>
The IP address that the `mongos` (page 1061) process will bind to and listen for connections. By default `mongos` (page 1061) listens for connections all interfaces. You may attach `mongos` (page 1061) to any interface; however, when attaching `mongos` (page 1061) to a publicly accessible interface ensure that you have implemented proper authentication and/or firewall restrictions to protect the integrity of your database.

command line option!–maxConns <number>

--maxConns <number>
Specifies the maximum number of simultaneous connections that `mongos` (page 1061) 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 1061) 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 1116), 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 1116) to a value higher than *20000*.

command line option!–objcheck

--objcheck
Forces the `mongos` (page 1061) 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.

command line option!–logpath <path>

--logpath <path>
Specify a path for the log file that will hold all diagnostic logging information.

Unless specified, `mongos` (page 1061) will output all log information to the standard output. Additionally, unless you also specify *--logappend* (page 1063), the logfile will be overwritten when the process restarts.

command line option!–logappend

--logappend
Specify to ensure that `mongos` (page 1061) appends additional logging data to the end of the logfile rather than overwriting the content of the log when the process restarts.

command line option!–setParameter <options>

--setParameter <options>
New in version 2.4.

Specifies an option to configure on startup. Specify multiple options with multiple *--setParameter* (page 1063) options. See *mongod Parameters* (page 1129) for full documentation of these parameters. The `setParameter` (page 894) database command provides access to many of these parameters. *--setParameter* (page 1063) supports the following options:

- `enableLocalhostAuthBypass` (page 1129)
- `enableTestCommands` (page 1129)
- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `supportCompatibilityFormPrivilegeDocuments` (page 1131)
- `syncdelay` (page 1131)
- `textSearchEnabled` (page 1132)

command line option!–syslog

--syslog
New in version 2.1.0.

Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with *--logpath* (page 1063).

> **Warning:** You cannot use *--syslog* (page 1063) with *--logpath* (page 1063).

command line option!–pidfilepath <path>

--pidfilepath <path>
Specify a file location to hold the "*PID*" or process ID of the `mongos` (page 1061) process. Useful for tracking the `mongos` (page 1061) process in combination with the *mongos --fork* option.

Without a specified *--pidfilepath* (page 1063) option, `mongos` (page 1061) creates no PID file.

command line option!–keyFile <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 1061) instances and components of the *sharded cluster*.

See also:

Sharded Cluster Security (page 505)

command line option!–nounixsocket

--nounixsocket
Disables listening on the UNIX socket. mongos (page 1061) always listens on the UNIX socket, unless *--nounixsocket* (page 1064) is set, *--bind_ip* (page 1062) is *not* set, or *--bind_ip* (page 1062) does *not* specify 127.0.0.1.

command line option!–unixSocketPrefix <path>

--unixSocketPrefix <path>
Specifies a path for the UNIX socket. Unless this option has a value mongos (page 1061) creates a socket with http://docs.mongodb.org/manualtmp as a prefix.

MongoDB will *always* create and listen on a UNIX socket, unless *--nounixsocket* (page 1064) is set, *--bind_ip* (page 1062) is *not* set, or *--bind_ip* (page 1062) specifies 127.0.0.1.

command line option!–fork

--fork
Enables a *daemon* mode for mongos (page 1061) 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.

command line option!–configdb <config1>,<config2><:port>,<config3>

--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: mongos (page 1061) instances read from the first *config server* in the list provided. All mongos (page 1061) instances **must** specify the hosts to the *--configdb* (page 1064) setting in the same order.

If your configuration databases reside in more that one data center, order the hosts in the *--configdb* (page 1064) argument so that the config database that is closest to the majority of your mongos (page 1061) instances is first servers in the list.

Warning: Never remove a config server from the *--configdb* (page 1064) parameter, even if the config server or servers are not available, or offline.

command line option!–test

--test
This option is for internal testing use only, and runs unit tests without starting a mongos (page 1061) instance.

command line option!–upgrade

--upgrade
This option updates the meta data format used by the *config database*.

command line option!–chunkSize <value>

--chunkSize <value>

The value of the *--chunkSize* (page 1064) determines the size of each *chunk*, *in megabytes*, of data distributed around the *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 538)" procedure if you need to change the chunk size on an existing sharded cluster.

command line option!–ipv6

--ipv6

Enables IPv6 support to allow clients to connect to mongos (page 1061) using IPv6 networks. MongoDB disables IPv6 support by default in mongod (page 1049) and all utilities.

command line option!–jsonp

--jsonp

Permits *JSONP* access via an HTTP interface. Consider the security implications of allowing this activity before enabling this option.

command line option!–noscripting

--noscripting

Disables the scripting engine.

command line option!–nohttpinterface

--nohttpinterface

New in version 2.1.2.

Disables the HTTP interface.

command line option!–localThreshold

--localThreshold

New in version 2.2.

--localThreshold (page 1065) affects the logic that mongos (page 1061) uses when selecting *replica set* members to pass read operations to from clients. Specify a value to *--localThreshold* (page 1065) in milliseconds. The default value is 15, which corresponds to the default value in all of the client *drivers* (page 575).

When mongos (page 1061) receives a request that permits reads to *secondary* members, the mongos (page 1061) 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 1065), mongos (page 1061) will construct the list of replica members that are within the latency allowed by this value.
- The mongos (page 1061) 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 1065) 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 1061) recalculates the average.

See the *Member Selection* (page 403) section of the *read preference* (page 398) documentation for more information.

command line option!–noAutoSplit

--noAutoSplit

New in version 2.0.7.

--noAutoSplit (page 1066) prevents mongos (page 1061) from automatically inserting metadata splits in a *sharded collection*. If set on all mongos (page 1061), this will prevent MongoDB from creating new chunks as the data in a collection grows.

Because any mongos (page 1061) in a cluster can create a split, to totally disable splitting in a cluster you must set *--noAutoSplit* (page 1066) on all mongos (page 1061).

> **Warning:** With *--noAutoSplit* (page 1066) enabled, the data in your sharded cluster may become imbalanced over time. Enable with caution.

SSL Options

See

Connect to MongoDB with SSL (page 179) for full documentation of MongoDB's support.

---

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>

New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, mongos (page 1061) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See userSource (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>

New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is MONGODB-CR, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, mongos (page 1061) also includes support for GSSAPI to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

mongo

Description

mongo

mongo (page 1066) is an interactive JavaScript shell interface to MongoDB, which provides a powerful interface for systems administrators as well as a way for developers to test queries and operations directly with the database. mongo (page 1066) also provides a fully functional JavaScript environment for use with a MongoDB. This document addresses the basic invocation of the mongo (page 1066) shell and an overview of its usage.

See also:

The mongo Shell (page 603)

Interface

Options

mongo

command line option!–shell

--shell
Enables the shell interface after evaluating a *JavaScript* file. If you invoke the `mongo` (page 1066) command and specify a JavaScript file as an argument, or use `--eval` (page 1067) to specify JavaScript on the command line, the `--shell` (page 1067) option provides the user with a shell prompt after the file finishes executing.

command line option!–nodb

--nodb
Prevents the shell from connecting to any database instances. Later, to connect to a database within the shell, see *Opening New Connections* (page 617).

command line option!–norc

--norc
Prevents the shell from sourcing and evaluating `~/.mongorc.js` on start up.

command line option!–quiet

--quiet
Silences output from the shell during the connection process.

command line option!–port <port>

--port `<port>`
Specifies the port where the `mongod` (page 1049) or `mongos` (page 1061) instance is listening. Unless specified `mongo` (page 1066) connects to `mongod` (page 1049) instances on port `27017`, which is the default `mongod` (page 1049) port.

command line option!–host <hostname>

--host `<hostname>`
specifies the host where the `mongod` (page 1049) or `mongos` (page 1061) is running to connect to as `<hostname>`. By default `mongo` (page 1066) will attempt to connect to a MongoDB process running on the localhost.

command line option!–eval <javascript>

--eval `<javascript>`
Evaluates a JavaScript expression specified as an argument to this option. `mongo` (page 1066) does not load its own environment when evaluating code: as a result many options of the shell environment are not available.

command line option!–username <username>, -u <username>

--username `<username>`, **-u** `<username>`
Specifies a username to authenticate to the MongoDB instance. Use in conjunction with the `--password` (page 1067) 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` (page 1066) will exit with an exception.

command line option!–password <password>, -p <password>

--password `<password>`, **-p** `<password>`
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the `--username` (page 1067) option to supply a username. If you specify a `--username` (page 1067) without the `--password` (page 1067) option, `mongo` (page 1066) will prompt for a password interactively, if the `mongod` (page 1049) or `mongos` (page 1061) requires authentication.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, mongo (page 1066) assumes that the database name specified in the *db address* (page 1069) holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See userSource (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is MONGODB-CR, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, mongo (page 1066) also includes support for GSSAPI to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–ssl

--ssl
Enable connection to a mongod (page 1049) or mongos (page 1061) that has SSL encryption.

command line option!–sslPEMKeyFile <filename>

--sslPEMKeyFile <filename>
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the .pem file that contains both the SSL certificate and key. Specify the file name of the .pem file using relative or absolute paths

Required when using the *--ssl* (page 1068) option if the mongod (page 1049) or mongos (page 1061) has sslCAFile (page 1128) enabled *without* sslWeakCertificateValidation (page 1128).

command line option!–sslPEMKeyPassword <value>

--sslPEMKeyPassword <value>
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the password to decrypt the root certificate chain specified by *--sslPEMKeyFile* (page 1068).

Only required if the certificate-key file is encrypted.

command line option!–sslCAFile <filename>

--sslCAFile <filename>
New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains the certificate from the Certificate Authority. Specify the file name of the `.pem` file using relative or absolute paths

command line option!–help, -h

`--help, -h`
Returns a basic help and usage text.

command line option!–version

`--version`
Returns the version of the shell.

command line option!–verbose

`--verbose`
Increases the verbosity of the output of the shell during the connection process.

command line option!–ipv6

`--ipv6`
Enables IPv6 support that allows `mongo` (page 1066) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongo` (page 1066), 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` (page 1066) 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 `--shell` (page 1067) option to return to a shell after the file finishes running.

Files `~/.dbshell`

`mongo` (page 1066) maintains a history of commands in the `.dbshell` file.

Note: `mongo` (page 1066) does not recorded interaction related to authentication in the history file, including `authenticate` and `db.addUser()` (page 993).

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` (page 1066) will read the `.mongorc.js` file from the home directory of the user invoking `mongo` (page 1066). In the file, users can define variables, customize the `mongo` (page 1066) shell prompt, or update information that they would like updated every time they launch a shell. If you use the shell to evaluate a JavaScript file or expression either on the command line with `--eval` (page 1067) or by specifying *a .js file to mongo* (page 1069), `mongo` (page 1066) will read the `.mongorc.js` file *after* the JavaScript has finished processing.

Specify the `--norc` (page 1067) option to disable reading `.mongorc.js`.

`http://docs.mongodb.org/manualtmp/mongo_edit<time_t>.js`

Created by `mongo` (page 1066) when editing a file. If the file exists `mongo` (page 1066) 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` (page 1066) 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 1070).

HOME

Specifies the path to the home directory where `mongo` (page 1066) will read the `.mongorc.js` file and write the `.dbshell` file.

HOMEDRIVE

On Windows systems, `HOMEDRIVE` (page 1070) specifies the path the directory where `mongo` (page 1066) will read the `.mongorc.js` file and write the `.dbshell` file.

HOMEPATH

Specifies the Windows path to the home directory where `mongo` (page 1066) will read the `.mongorc.js` file and write the `.dbshell` file.

Keyboard Shortcuts

The `mongo` (page 1066) shell supports the following keyboard shortcuts: [1]

Keybinding	Function
Up arrow	Retrieve previous command from history
Down-arrow	Retrieve next command from history
Home	Go to beginning of the line
End	Go to end of the line
Tab	Autocomplete method/command
Left-arrow	Go backward one character
Right-arrow	Go forward one character
Ctrl-left-arrow	Go backward one word
Ctrl-right-arrow	Go forward one word
Meta-left-arrow	Go backward one word
Meta-right-arrow	Go forward one word
Ctrl-A	Go to the beginning of the line
Ctrl-B	Go backward one character
Ctrl-C	Exit the `mongo` (page 1066) shell
Ctrl-D	Delete a char (or exit the `mongo` (page 1066) shell)
Ctrl-E	Go to the end of the line
Ctrl-F	Go forward one character

Continued on next page

[1] MongoDB accommodates multiple keybinding. Since 2.0, `mongo` (page 1066) includes support for basic emacs keybindings.

Table 56.1 – continued from previous page

Keybinding	Function
Ctrl-G	Abort
Ctrl-J	Accept/evaluate the line
Ctrl-K	Kill/erase the line
Ctrl-L or type `cls`	Clear the screen
Ctrl-M	Accept/evaluate the line
Ctrl-N	Retrieve next command from history
Ctrl-P	Retrieve previous command from history
Ctrl-R	Reverse-search command history
Ctrl-S	Forward-search command history
Ctrl-T	Transpose characters
Ctrl-U	Perform Unix line-discard
Ctrl-W	Perform Unix word-rubout
Ctrl-Y	Yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H	Backward-delete a character
Ctrl-I	Complete, same as Tab
Meta-B	Go backward one word
Meta-C	Capitalize word
Meta-D	Kill word
Meta-F	Go forward one word
Meta-L	Change word to lowercase
Meta-U	Change word to uppercase
Meta-Y	Yank-pop
Meta-Backspace	Backward-kill word
Meta-<	Retrieve the first command in command history
Meta->	Retrieve the last command in command history

Use

Typically users invoke the shell with the `mongo` (page 1066) 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 1067) 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`* 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.

56.1.2 Windows Services

The `mongod.exe` (page 1072) and `mongos.exe` (page 1073) describe the options available for configuring MongoDB when running as a Windows Service. The `mongod.exe` (page 1072) and `mongos.exe` (page 1073) binaries provide a superset of the `mongod` (page 1049) and `mongos` (page 1061) options.

mongod.exe

Synopsis

`mongod.exe` (page 1072) is the build of the MongoDB daemon (i.e. `mongod` (page 1049)) for the Windows platform. `mongod.exe` (page 1072) has all of the features of `mongod` (page 1049) on Unix-like platforms and is completely compatible with the other builds of `mongod` (page 1049). In addition, `mongod.exe` (page 1072) provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongod.exe` (page 1072). All `mongod` (page 1049) options are available. See the "*mongod* (page 1049)" and the "*Configuration File Options* (page 1115)" documents for more information regarding `mongod.exe` (page 1072).

To install and use `mongod.exe` (page 1072), read the "*Install MongoDB on Windows* (page 16)" document.

Options

mongod.**exe**

mongod.**exe**

command line option!–install

--install
Installs `mongod.exe` (page 1072) as a Windows Service and exits.

command line option!–remove

--remove
Removes the `mongod.exe` (page 1072) Windows Service. If `mongod.exe` (page 1072) is running, this operation will stop and then remove the service.

Note: *`--remove`* (page 1073) requires the *`--serviceName`* (page 1074) if you configured a non-default *`--serviceName`* (page 1074) during the *`--install`* (page 1073) operation.

command line option!–reinstall

--reinstall
Removes `mongod.exe` (page 1072) and reinstalls `mongod.exe` (page 1072) as a Windows Service.

command line option!–serviceName <name>

--serviceName <name>
Default: "MongoDB"

Set the service name of `mongod.exe` (page 1072) when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *`--serviceName`* (page 1074) in conjunction with either the *`--install`* (page 1073) or *`--remove`* (page 1073) install option.

command line option!–serviceDisplayName <name>

--serviceDisplayName <name>
Default: "Mongo DB"

Sets the name listed for MongoDB on the Services administrative application.

command line option!–serviceDescription <description>

--serviceDescription <description>
Default: "MongoDB Server"

Sets the `mongod.exe` (page 1072) service description.

You must use *--serviceDescription* (page 1074) in conjunction with the *--install* (page 1073) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

command line option!–serviceUser <user>

--serviceUser <user>
Runs the `mongod.exe` (page 1072) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 1074) in conjunction with the *--install* (page 1073) option.

command line option!–servicePassword <password>

--servicePassword <password>
Sets the password for <user> for `mongod.exe` (page 1072) when running with the *--serviceUser* (page 1074) option.

You must use *--servicePassword* (page 1074) in conjunction with the *--install* (page 1073) option.

mongos.exe

Synopsis

`mongos.exe` (page 1073) is the build of the MongoDB Shard (i.e. `mongos` (page 1061)) for the Windows platform. `mongos.exe` (page 1073) has all of the features of `mongos` (page 1061) on Unix-like platforms and is completely compatible with the other builds of `mongos` (page 1061). In addition, `mongos.exe` (page 1073) provides several options for interacting with the Windows platform itself.

This document only references options that are unique to `mongos.exe` (page 1073). All `mongos` (page 1061) options are available. See the "*mongos* (page 1061)" and the "*Configuration File Options* (page 1115)" documents for more information regarding `mongos.exe` (page 1073).

To install and use `mongos.exe` (page 1073), read the "*Install MongoDB on Windows* (page 16)" document.

Options

mongos.**exe**

mongos.**exe**

command line option!–install

--install
Installs `mongos.exe` (page 1073) as a Windows Service and exits.

command line option!–remove

--remove

Removes the mongos.exe (page 1073) Windows Service. If mongos.exe (page 1073) is running, this operation will stop and then remove the service.

Note: *--remove* (page 1073) requires the *--serviceName* (page 1074) if you configured a non-default *--serviceName* (page 1074) during the *--install* (page 1073) operation.

command line option!–reinstall

--reinstall

Removes mongos.exe (page 1073) and reinstalls mongos.exe (page 1073) as a Windows Service.

command line option!–serviceName <name>

--serviceName <name>

Default: "MongoS"

Set the service name of mongos.exe (page 1073) when running as a Windows Service. Use this name with the `net start <name>` and `net stop <name>` operations.

You must use *--serviceName* (page 1074) in conjunction with either the *--install* (page 1073) or *--remove* (page 1073) install option.

command line option!–serviceDisplayName <name>

--serviceDisplayName <name>

Default: "Mongo DB Router"

Sets the name listed for MongoDB on the Services administrative application.

command line option!–serviceDescription <description>

--serviceDescription <description>

Default: "Mongo DB Sharding Router"

Sets the mongos.exe (page 1073) service description.

You must use *--serviceDescription* (page 1074) in conjunction with the *--install* (page 1073) option.

Note: For descriptions that contain spaces, you must enclose the description in quotes.

command line option!–serviceUser <user>

--serviceUser <user>

Runs the mongos.exe (page 1073) service in the context of a certain user. This user must have "Log on as a service" privileges.

You must use *--serviceUser* (page 1074) in conjunction with the *--install* (page 1073) option.

command line option!–servicePassword <password>

--servicePassword <password>

Sets the password for `<user>` for mongos.exe (page 1073) when running with the *--serviceUser* (page 1074) option.

You must use *--servicePassword* (page 1074) in conjunction with the *--install* (page 1073) option.

56.1.3 Binary Import and Export Tools

mongodump (page 1075) provides a method for creating *BSON* dump files from the mongod (page 1049) instances, while mongorestore (page 1079) makes it possible to restore these dumps. bsondump (page 1084) converts BSON dump files into *JSON*. The mongooplog (page 1086) utility provides the ability to stream *oplog* entries outside of normal replication.

mongodump

Synopsis

mongodump (page 1075) 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 133). Use mongodump (page 1075) in conjunction with mongorestore (page 1079) to restore databases.

mongodump (page 1075) can read data from either *mongod* or mongos (page 1061) instances, in addition to reading directly from MongoDB data files without an active mongod (page 1049).

Important: mongodump (page 1075) does *not* create output for the `local` database.

Note: The format of data created by mongodump (page 1075) tool from the 2.2 distribution or later is different and incompatible with earlier versions of mongod (page 1049).

See also:

mongorestore (page 1079), *Create Backup of a Sharded Cluster with Database Dumps* (page 148) and *Backup Strategies for MongoDB Systems* (page 133).

Options

mongodump

mongodump

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the mongodump (page 1075) utility and exits.

command line option!–host <hostname><:port>

--host `<hostname><:port>`
Specifies a resolvable hostname for the mongod (page 1049) that you wish to use to create the database dump. By default mongodump (page 1075) 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* argument with a setname, followed by a slash and a comma-separated list of host names and port numbers. The mongodump (page 1075) 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.

command line option!–port <port>

--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* option.

command line option!–ipv6

--ipv6
Enables IPv6 support that allows mongodump (page 1075) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongodump (page 1075), disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.4: MongoDB added support for SSL connections to mongod (page 1049) instances in mongodump.

Note: SSL support in mongodump is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongodump does not support connections to mongod (page 1049) instances that require client certificate validation.

Allows mongodump (page 1075) to connect to mongod (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, mongodump (page 1075) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, mongodump (page 1075) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See userSource (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is MONGODB-CR, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, mongodump (page 1075) also includes support for GSSAPI to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 1081) option enables mongodump (page 1075) to attach directly to local data files and copy the data without the mongod (page 1049). To run with *--dbpath* (page 1081), mongodump (page 1075) needs to restrict access to the data directory: as a result, no mongod (page 1049) can access the same path while the process runs.

command line option!–directoryperdb

--directoryperdb
Use the *--directoryperdb* (page 1081) in conjunction with the corresponding option to mongod (page 1049). This option allows mongodump (page 1075) to read data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 1081) option.

command line option!–journal

--journal
Allows mongodump (page 1075) 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 1081) option.

command line option!–db <db>, -d <db>

--db <db>, **-d** <db>
Use the *--db* (page 1081) option to specify a database for mongodump (page 1075) to backup. If you do not specify a DB, mongodump (page 1075) copies all databases in this instance into the dump files. Use this option to backup or copy a smaller subset of your data.

command line option!–collection <collection>, -c <collection>

--collection <collection>, **-c** <collection>
Use the *--collection* (page 1082) option to specify a collection for mongodump (page 1075) 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.

command line option!–out <path>, -o <path>

--out <path>, **-o** <path>
Specifies a directory where mongodump (page 1075) saves the output of the database dump. By default, mongodump (page 1075) saves output files in a directory named dump in the current working directory.

To send the database dump to standard output, specify "–" instead of a path. Write to standard output if you want process the output before saving it, such as to use gzip to compress the dump. When writing standard output, mongodump (page 1075) does not write the metadata that writes in a <dbname>.metadata.json file when writing to files directly.

command line option!–query <json>, -q <json>

--query <json>, **-q** <json>
Provides a query to limit (optionally) the documents included in the output of mongodump (page 1075).

command line option!–oplog

--oplog

Use this option to ensure that `mongodump` (page 1075) creates a dump of the database that includes an *oplog*, to create a point-in-time snapshot of the state of a `mongod` (page 1049) instance. To restore to a specific point-in-time backup, use the output created with this option in conjunction with *mongorestore --oplogReplay*.

Without *--oplog* (page 1078), 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 1078) has no effect when running `mongodump` (page 1075) against a `mongos` (page 1061) instance to dump the entire contents of a sharded cluster. However, you can use *--oplog* (page 1078) to dump individual shards.

Note: *--oplog* (page 1078) only works against nodes that maintain an *oplog*. This includes all members of a replica set, as well as *master* nodes in master/slave replication deployments.

command line option!–repair

--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` (page 1049) crash.

command line option!–forceTableScan

--forceTableScan

Forces `mongodump` (page 1075) to scan the data store directly: typically, `mongodump` (page 1075) saves entries as they appear in the index of the `_id` field. Use *--forceTableScan* (page 1096) 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 1096), `mongodump` (page 1075) does not use `$snapshot` (page 832). As a result, the dump produced by `mongodump` (page 1075) can reflect the state of the database at many different points in time.

Warning: Use *--forceTableScan* (page 1096) with extreme caution and consideration.

Warning: Changed in version 2.2: When used in combination with `fsync` (page 888) or `db.fsyncLock()` (page 1004), `mongod` (page 1049) may block some reads, including those from `mongodump` (page 1075), when queued write operation waits behind the `fsync` (page 888) lock.

Behavior

When running `mongodump` (page 1075) against a `mongos` (page 1061) 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 *Use mongodump and mongorestore to Backup and Restore MongoDB Databases* (page 135) for a larger overview of `mongodump` (page 1075) usage. Also see the "*mongorestore* (page 1079)" document for an overview of the `mongorestore` (page 1079), 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` (page 1075) creates a backup of the database instance stored in the `http://docs.mongodb.org/manualsrv/mongodb` directory on the local machine. This requires that no `mongod` (page 1049) instance is using the `http://docs.mongodb.org/manualsrv/mongodb` directory.

```
mongodump --dbpath /srv/mongodb
```

In the final example, `mongodump` (page 1075) creates a database dump located at `http://docs.mongodb.org/manualopt/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 --out /opt/backup/
```

mongorestore

Synopsis

The `mongorestore` (page 1079) program writes data from a binary database dump created by `mongodump` (page 1075) to a MongoDB instance. `mongorestore` (page 1079) can create a new database or add data to an existing database.

`mongorestore` (page 1079) can write data to either *mongod* or `mongos` (page 1061) instances, in addition to writing directly to MongoDB data files without an active `mongod` (page 1049).

If you restore to an existing database, `mongorestore` (page 1079) will only insert into the existing database, and does not perform updates of any kind. If existing documents have the same value `_id` field in the target database and collection, `mongorestore` (page 1079) will *not* overwrite those documents.

Remember the following properties of `mongorestore` (page 1079) behavior:

- `mongorestore` (page 1079) recreates indexes recorded by `mongodump` (page 1075).
- all operations are inserts, not updates.
- `mongorestore` (page 1079) does not wait for a response from a `mongod` (page 1049) to ensure that the MongoDB process has received or recorded the operation.

  The `mongod` (page 1049) will record any errors to its log that occur during a restore operation, but `mongorestore` (page 1079) will not receive errors.

Note: The format of data created by `mongodump` (page 1075) tool from the 2.2 distribution or later is different and incompatible with earlier versions of `mongod` (page 1049).

Options

mongorestore

mongorestore

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`).

command line option!–version

--version
Returns the version of the `mongorestore` (page 1079) tool.

command line option!–host <hostname><:port>

--host <hostname><:port>
Specifies a resolvable hostname for the `mongod` (page 1049) to which you want to restore the database. By default `mongorestore` (page 1079) will attempt to connect to a MongoDB process running on the localhost port number `27017`. For an example of *--host* (page 1067), see *Restore a Database with mongorestore* (page 137).

Optionally, specify a port number to connect a MongoDB instance running on a port other than `27017`.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–port <port>

--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 1067) command. For an example of *--port* (page 1067), see *Restore a Database with mongorestore* (page 137).

command line option!–ipv6

--ipv6
Enables IPv6 support that allows `mongorestore` (page 1079) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongorestore` (page 1079), disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.4: MongoDB added support for SSL connections to `mongod` (page 1049) instances in mongorestore.

Note: SSL support in mongorestore is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongorestore does not support connections to `mongod` (page 1049) instances that require client certificate validation.

Allows `mongorestore` (page 1079) to connect to `mongod` (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--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 1067) option to supply a password. For an example of *--username* (page 1067), see *Restore a Database with mongorestore* (page 137).

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* (page 1067) option to supply a username. For an example of *--password* (page 1067), see *Restore a Database with mongorestore* (page 137).

If you specify a *--username* (page 1067) without the *--password* (page 1067) option, mongorestore (page 1079) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, mongorestore (page 1079) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See userSource (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is MONGODB-CR, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, mongorestore (page 1079) also includes support for GSSAPI to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 1081) option enables mongorestore (page 1079) to attach directly to local data files and insert the data without the mongod (page 1049). To run with *--dbpath* (page 1081), mongorestore (page 1079) needs to lock access to the data directory: as a result, no mongod (page 1049) can access the same path while the process runs. For an example of *--dbpath* (page 1081), see *Restore without a Running mongod* (page 137).

command line option!–directoryperdb

--directoryperdb
Use the *--directoryperdb* (page 1081) in conjunction with the corresponding option to mongod (page 1049), which allows mongorestore (page 1079) 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 1081) option.

command line option!–journal

--journal
Allows mongorestore (page 1079) 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 1081) option. For an example of *--journal* (page 1081), see *Restore without a Running mongod* (page 137).

command line option!–db <db>, -d <db>

--db <db>, **-d** <db>
Use the *--db* (page 1081) option to specify a database for mongorestore (page 1079) to restore data *into*. If the database doesn't exist, mongorestore (page 1079) will create the specified database. If you do not

specify a `<db>`, `mongorestore` (page 1079) 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 1081) does *not* control which *BSON* files `mongorestore` (page 1079) restores. You must use the `mongorestore` (page 1079) *path option* (page 1083) to limit that restored data.

command line option!–collection <collection>, -c <collection>

--collection <collection>**, -c** <collection>

Use the *--collection* (page 1082) option to specify a collection for `mongorestore` (page 1079) to restore. If you do not specify a `<collection>`, `mongorestore` (page 1079) 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.

command line option!–objcheck

--objcheck

Forces the `mongorestore` (page 1079) to validate all requests from clients upon receipt to ensure that clients never insert invalid documents into the database. For objects with a high degree of sub-document nesting, *--objcheck* (page 1107) can have a small impact on performance. You can set *--noobjcheck* (page 1082) to disable object checking at run-time.

Changed in version 2.4: MongoDB enables *--objcheck* (page 1107) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–noobjcheck

--noobjcheck

New in version 2.4.

Disables the default document validation that MongoDB performs on all incoming BSON documents.

command line option!–filter '<JSON>'

--filter '<JSON>'

Limits the documents that `mongorestore` (page 1079) 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. For an example of *--filter* (page 1082), see *Restore a Subset of data from a Binary Database Dump* (page 137).

command line option!–drop

--drop

Modifies the restoration procedure to drop every collection from the target database before restoring the collection from the dumped backup.

command line option!–oplogReplay

--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*" command. For an example of *--oplogReplay* (page 1082), see *Restore Point in Time Oplog Backup* (page 137).

command line option!–keepIndexVersion

--keepIndexVersion

Prevents `mongorestore` (page 1079) from upgrading the index to the latest version during the restoration process.

command line option!–w <number of replicas per write>

--w <number of replicas per write>
New in version 2.2.

Specifies the *write concern* for each write operation that mongorestore (page 1079) writes to the target database. By default, mongorestore (page 1079) does not wait for a response for *write acknowledgment* (page 396).

command line option!–noOptionsRestore

--noOptionsRestore
New in version 2.2.

Prevents mongorestore (page 1079) from setting the collection options, such as those specified by the collMod (page 892) *database command*, on restored collections.

command line option!–noIndexRestore

--noIndexRestore
New in version 2.2.

Prevents mongorestore (page 1079) from restoring and building indexes as specified in the corresponding mongodump (page 1075) output.

command line option!–oplogLimit <timestamp>

--oplogLimit <timestamp>
New in version 2.2.

Prevents mongorestore (page 1079) 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 1083) in conjunction with the *--oplogReplay* (page 1082) option.

<path>
The final argument of the mongorestore (page 1079) command is a directory path. This argument specifies the location of the database dump from which to restore.

Usage

See *Use mongodump and mongorestore to Backup and Restore MongoDB Databases* (page 135) for a larger overview of mongorestore (page 1079) usage. Also see the "*mongodump* (page 1075)" document for an overview of the mongodump (page 1075), which provides the related inverse functionality.

Consider the following example:

```
mongorestore --collection people --db accounts dump/accounts/people.bson
```

Here, mongorestore (page 1079) 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 (page 1079) restores data to the instance running on the localhost interface on port `27017`.

In the next example, mongorestore (page 1079) restores a backup of the database instance located in `dump` to a database instance stored in the `http://docs.mongodb.org/manualsrv/mongodb` on the local machine. This requires that there are no active mongod (page 1049) instances attached to `http://docs.mongodb.org/manualsrv/mongodb` data directory.

```
mongorestore --dbpath /srv/mongodb
```

In the final example, `mongorestore` (page 1079) restores a database dump located at `http://docs.mongodb.org/manualopt/backup/mongodump-2011-10-24`, from a database running on port `37017` on the host `mongodb1.example.net`. `mongorestore` (page 1079) 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/mon
```

bsondump

Synopsis

The `bsondump` (page 1084) converts *BSON* files into human-readable formats, including *JSON*. For example, `bsondump` (page 1084) is useful for reading the output files generated by `mongodump` (page 1075).

Important: `bsondump` (page 1084) is a diagnostic tool for inspecting BSON files, not a tool for data ingestion or other application use.

Options

bsondump

bsondump

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the `bsondump` (page 1084) utility.

command line option!–objcheck

--objcheck
Validates each *BSON* object before outputting it in *JSON* format. By default, `bsondump` (page 1084) enables *`--objcheck`* (page 1107). For objects with a high degree of sub-document nesting, *`--objcheck`* (page 1107) can have a small impact on performance. You can set *`--noobjcheck`* (page 1082) to disable object checking.

Changed in version 2.4: MongoDB enables *`--objcheck`* (page 1107) by default, to prevent any client from inserting malformed or invalid BSON into a MongoDB database.

command line option!–noobjcheck

--noobjcheck
New in version 2.4.

Disables the default document validation that `bsondump` (page 1084) performs on all BSON documents.

command line option!–filter '<JSON>'

--filter '<JSON>'

Limits the documents that bsondump (page 1084) 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.

command line option!–type <=json|=debug>

--type <=json|=debug>

Changes the operation of bsondump (page 1084) from outputting "*JSON*" (the default) to a debugging format.

<bsonfilename>

The final argument to bsondump (page 1084) is a document containing *BSON*. This data is typically generated by mongodump (page 1075) or by MongoDB in a *rollback* operation.

Usage

By default, bsondump (page 1084) 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 2.2.

Synopsis

mongooplog (page 1086) 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 (page 1049) 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* host running during the migration, consider using mongodump (page 1075) and mongorestore (page 1079) or another *backup* (page 133) operation, which may be better suited to your operation.

Note: If the mongod (page 1049) instance specified by the *--from* argument is running with authentication (page 1118), then mongooplog (page 1086) will not be able to copy oplog entries.

See also:

mongodump (page 1075), mongorestore (page 1079), "*Backup Strategies for MongoDB Systems* (page 133)", "*Replica Set Oplog* (page 405).

Options

mongooplog

mongooplog

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the `mongooplog` (page 1086) utility.

command line option!–host <hostname><:port>, -h

--host `<hostname><:port>`**, -h**
Specifies a resolvable hostname for the `mongod` (page 1049) instance to which `mongooplog` (page 1086) will apply *oplog* operations retrieved from the serve specified by the `--from` option.

`mongooplog` (page 1086) assumes that all target `mongod` (page 1049) 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` (page 1049) instance by specifying the host and port number directly.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–port

--port
Specifies the port number of the `mongod` (page 1049) instance where `mongooplog` (page 1086) 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` command.

command line option!–ipv6

--ipv6
Enables IPv6 support that allows `mongooplog` (page 1086) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongooplog` (page 1086), disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.4: MongoDB added support for SSL connections to `mongod` (page 1049) instances in mongooplog.

Note: SSL support in mongooplog is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongooplog does not support connections to `mongod` (page 1049) instances that require client certificate validation.

Allows `mongooplog` (page 1086) to connect to `mongod` (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *--password* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, `mongooplog` (page 1086) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, `mongooplog` (page 1086) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See `userSource` (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is `MONGODB-CR`, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, `mongooplog` (page 1086) also includes support for `GSSAPI` to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>
Specifies a directory, containing MongoDB data files, to which `mongooplog` (page 1086) will apply operations from the *oplog* of the database specified with the *--from* option. When used, the *--dbpath* (page 1081) option enables `mongo` (page 1066) to attach directly to local data files and write data without a running `mongod` (page 1049) instance. To run with *--dbpath* (page 1081), `mongooplog` (page 1086) needs to restrict access to the data directory: as a result, no `mongod` (page 1049) can be access the same path while the process runs.

command line option!–directoryperdb

--directoryperdb
Use the *--directoryperdb* (page 1081) in conjunction with the corresponding option to `mongod` (page 1049). This option allows `mongooplog` (page 1086) to write to data files organized with each database located in a distinct directory. This option is only relevant when specifying the *--dbpath* (page 1081) option.

command line option!–journal

--journal
Allows `mongooplog` (page 1086) 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 1081) option.

command line option!–seconds <number>, -s <number>

--seconds <number>, **-s** <number>
Specify a number of seconds of operations for `mongooplog` (page 1086) to pull from the *remote host*. Unless specified the default value is `86400` seconds, or 24 hours.

command line option!–from <host[:port]>

--from <host[:port]>
Specify the host for `mongooplog` (page 1086) to retrieve *oplog* operations from. `mongooplog` (page 1086) *requires* this option.

Unless you specify the *--host* option, `mongooplog` (page 1086) will apply the operations collected with this option to the oplog of the `mongod` (page 1049) instance running on the localhost interface connected to port `27017`.

command line option!–oplogns <namespace>

--oplogns <namespace>
Specify a namespace in the *--from* 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` (page 1086) command:

```
mongooplog  --from mongodb0.example.net --host mongodb1.example.net
```

Here, entries from the *oplog* of the `mongod` (page 1049) running on port `27017`. This only pull entries from the last 24 hours.

Use the *--seconds* 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` (page 1086) 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` (page 1086) migrates entries to the `mongod` (page 1049) process running on the localhost interface connected to the `27017` port. `mongooplog` (page 1086) can also operate directly on MongoDB's data files if no `mongod` (page 1049) is running on the *target* host. Consider the following example:

```
mongooplog  --from mongodb0.example.net --dbpath /srv/mongodb --journal
```

Here, `mongooplog` (page 1086) imports *oplog* operations from the `mongod` (page 1049) host connected to port `27017`. This migrates operations to the MongoDB data files stored in the `http://docs.mongodb.org/manualsrv/mongodb` directory. Additionally `mongooplog` (page 1086) will use the durability *journal* to ensure that the data files remain in a consistent state.

56.1.4 Data Import and Export Tools

mongoimport (page 1089) provides a method for taking data in *JSON*, *CSV*, or *TSV* and importing it into a mongod (page 1049) instance. mongoexport (page 1093) provides a method to export data from a mongod (page 1049) instance into JSON, CSV, or TSV.

Note: The conversion between BSON and other formats lacks full type fidelity. Therefore you cannot use mongoimport (page 1089) and mongoexport (page 1093) for round-trip import and export operations.

mongoimport

Synopsis

The mongoimport (page 1089) tool provides a route to import content from a JSON, CSV, or TSV export created by mongoexport (page 1093), or potentially, another third-party export tool. See the "*Import and Export MongoDB Data* (page 166)" document for a more in depth usage overview, and the "*mongoexport* (page 1093)" document for more information regarding mongoexport (page 1093), which provides the inverse "importing" capability.

Note: Do not use mongoimport (page 1089) and mongoexport (page 1093) for full instance, production backups because they will not reliably capture data type information. Use mongodump (page 1075) and mongorestore (page 1079) as described in "*Backup Strategies for MongoDB Systems* (page 133)" for this kind of functionality.

Options

mongoimport

mongoimport

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the mongoimport (page 1089) program.

command line option!–host <hostname><:port>, -h

--host <hostname><:port>**, -h**
Specifies a resolvable hostname for the mongod (page 1049) to which you want to restore the database. By default mongoimport (page 1089) 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 1067) argument with a setname, followed by a slash and a comma-separated list of host and port names. mongoimport (page 1089) will, given the seed of at least one connected set member, connect to the *primary* 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.

command line option!–port <port>

--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* command.

command line option!–ipv6

--ipv6
Enables IPv6 support that allows `mongoimport` (page 1089) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongoimport` (page 1089), disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.4: MongoDB added support for SSL connections to `mongod` (page 1049) instances in mongoimport.

Note: SSL support in mongoimport is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongoimport does not support connections to `mongod` (page 1049) instances that require client certificate validation.

Allows `mongoimport` (page 1089) to connect to `mongod` (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--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* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongoimport --username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, `mongoimport` (page 1089) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, `mongoimport` (page 1089) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See `userSource` (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is `MONGODB-CR`, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, `mongoimport` (page 1089) also includes support for `GSSAPI` to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>

Specifies the directory of the MongoDB data files. If used, the *--dbpath* option enables `mongoimport` (page 1089) to attach directly to local data files and insert the data without the `mongod` (page 1049). To run with `--dbpath`, `mongoimport` (page 1089) needs to lock access to the data directory: as a result, no `mongod` (page 1049) can access the same path while the process runs.

command line option!–directoryperdb

--directoryperdb

Use the *--directoryperdb* (page 1081) in conjunction with the corresponding option to `mongod` (page 1049), which allows `mongoimport` (page 1089) 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 1081) option.

command line option!–journal

--journal

Allows `mongoexport` (page 1093) 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 1081) option.

command line option!–db <db>, -d <db>

--db <db>**, -d** <db>

Use the *--db* (page 1081) option to specify a database for `mongoimport` (page 1089) to import data.

command line option!–collection <collection>, -c <collection>

--collection <collection>**, -c** <collection>

Use the *--collection* (page 1082) option to specify a collection for `mongoimport` (page 1089) to import.

command line option!–fields <field1<,field2>>, -f <field1[,field2]>

--fields <field1<,field2>>**, -f** <field1[,field2]>

Specify a comma separated list of field names when importing *csv* or *tsv* files that do not have field names in the first (i.e. header) line of the file.

command line option!–fieldFile <filename>

--fieldFile <filename>

As an alternative to *--fields* (page 1091) the *--fieldFile* (page 1091) option allows you to specify a file (e.g. `<file>`) to that holds a list of field names if your *csv* or *tsv* file does not include field names in the first (i.e. header) line of the file. Place one field per line.

command line option!–ignoreBlanks

--ignoreBlanks

In *csv* and *tsv* exports, ignore empty fields. If not specified, `mongoimport` (page 1089) creates fields without values in imported documents.

command line option!–type <json|csv|tsv>

--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.

command line option!–file <filename>

--file <filename>
Specify the location of a file containing the data to import. `mongoimport` (page 1089) will read data from standard input (e.g. "stdin.") if you do not specify a file.

command line option!–drop

--drop
Modifies the importation procedure so that the target instance drops every collection before restoring the collection from the dumped backup.

command line option!–headerline

--headerline
If using "*--type csv*" or "*--type tsv*," use the first line as field names. Otherwise, `mongoimport` (page 1089) will import the first line as a distinct document.

command line option!–upsert

--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 1092) `mongoimport` (page 1089) will upsert on the basis of the `_id` field.

command line option!–upsertFields <field1[,field2]>

--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.

command line option!–stopOnError

--stopOnError
New in version 2.2.

Forces `mongoimport` (page 1089) to halt the import operation at the first error rather than continuing the operation despite errors.

command line option!–jsonArray

--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* to import data written as a single *JSON* array. Limited to imports of 16 MB or smaller.

Usage

In this example, `mongoimport` (page 1089) imports the *csv* formatted data in the `http://docs.mongodb.org/manualopt/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 (page 1089) 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 (page 1089) 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/manualsrv/mongodb/`. if the import process encounters an error, the mongoimport (page 1089) will halt because of the *--stopOnError* option.

```
mongoimport --db sales --collection contacts --stopOnError --dbpath /srv/mongodb/
```

In the final example, mongoimport (page 1089) imports data from the file `http://docs.mongodb.org/manualopt/backups/mdb1-examplenet.json` into the collection `contacts` within the database `marketing` on a remote MongoDB database. This mongoimport (page 1089) accesses the mongod (page 1049) 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 (page 1093) is a utility that produces a JSON or CSV export of data stored in a MongoDB instance. See the "*Import and Export MongoDB Data* (page 166)" document for a more in depth usage overview, and the "*mongoimport* (page 1089)" document for more information regarding the mongoimport (page 1089) utility, which provides the inverse "importing" capability.

Note: Do not use mongoimport (page 1089) and mongoexport (page 1093) for full-scale backups because they may not reliably capture data type information. Use mongodump (page 1075) and mongorestore (page 1079) as described in "*Backup Strategies for MongoDB Systems* (page 133)" for this kind of functionality.

Options

mongoexport

mongoexport

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the mongoexport (page 1093) utility.

command line option!–host <hostname><:port>

--host <hostname><:port>
Specifies a resolvable hostname for the mongod (page 1049) from which you want to export data. By default mongoexport (page 1093) 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.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–port <port>

--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* command.

command line option!–ipv6

--ipv6
Enables IPv6 support that allows mongoexport (page 1093) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongoexport (page 1093), disable IPv6 support by default.

command line option!–ssl

--ssl
New in version 2.4: MongoDB added support for SSL connections to mongod (page 1049) instances in mongoexport.

Note: SSL support in mongoexport is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongoexport does not support connections to mongod (page 1049) instances that require client certificate validation.

Allows mongoexport (page 1093) to connect to mongod (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--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* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, mongoexport (page 1093) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g `--username`) credentials.

By default, `mongoexport` (page 1093) assumes that the database specified to the `--db` (page 1081) argument holds the user's credentials, unless you specify `--authenticationDatabase` (page 1067).

See `userSource` (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is `MONGODB-CR`, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, `mongoexport` (page 1093) also includes support for `GSSAPI` to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the `--dbpath` option enables `mongoexport` (page 1093) to attach directly to local data files and insert the data without the `mongod` (page 1049). To run with `--dbpath`, `mongoexport` (page 1093) needs to lock access to the data directory: as a result, no `mongod` (page 1049) can access the same path while the process runs.

command line option!–directoryperdb

--directoryperdb
Use the `--directoryperdb` (page 1081) in conjunction with the corresponding option to `mongod` (page 1049), which allows `mongoexport` (page 1093) 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 1081) option.

command line option!–journal

--journal
Allows `mongoexport` (page 1093) 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 1081) option.

command line option!–db <db>, -d <db>

--db <db>, **-d** <db>
Use the `--db` (page 1081) option to specify the name of the database that contains the collection you want to export.

command line option!–collection <collection>, -c <collection>

--collection <collection>, **-c** <collection>
Use the `--collection` (page 1082) option to specify the collection that you want `mongoexport` (page 1093) to export.

command line option!–fields <field1[,field2]>, -f <field1[,field2]>

--fields <field1[,field2]>, **-f** <field1[,field2]>
Specify a field or fields to *include* in the export. Use a comma separated list of fields to specify multiple fields.

For `--csv` output formats, `mongoexport` (page 1093) includes only the specified field(s), and the specified field(s) can be a field within a sub-document.

For *JSON* output formats, `mongoexport` (page 1093) includes only the specified field(s) **and** the `_id` field, and if the specified field(s) is a field within a sub-document, the `mongoexport` (page 1093) includes the sub-document with all its fields, not just the specified field within the document.

command line option!–fieldFile <file>

--fieldFile `<file>`
As an alternative to *--fields*, the *--fieldFile* option allows you to specify in a file the field or fields to *include* in the export and is **only valid** with the *--csv* option. The file must have only one field per line, and the line(s) must end with the LF character (`0x0A`).

`mongoexport` (page 1093) includes only the specified field(s). The specified field(s) can be a field within a sub-document.

command line option!–query <JSON>

--query `<JSON>`
Provides a *JSON document* as a query that optionally limits the documents returned in the export.

command line option!–csv

--csv
Changes the export format to a comma separated values (CSV) format. By default `mongoexport` (page 1093) writes data using one *JSON* document for every MongoDB document.

If you specify *--csv* (page 1096), then you must also use either the *--fields* (page 1091) or the *--fieldFile* (page 1091) option to declare the fields to export from the collection.

command line option!–jsonArray

--jsonArray
Modifies the output of `mongoexport` (page 1093) to write the entire contents of the export as a single *JSON* array. By default `mongoexport` (page 1093) writes data using one JSON document for every MongoDB document.

command line option!–slaveOk, -k

--slaveOk, -k
Allows `mongoexport` (page 1093) to read data from secondary or slave nodes when using `mongoexport` (page 1093) with a replica set. This option is only available if connected to a `mongod` (page 1049) or `mongos` (page 1061) and is not available when used with the "*mongoexport --dbpath*" option.

This is the default behavior.

command line option!–out <file>, -o <file>

--out `<file>`, **-o** `<file>`
Specify a file to write the export to. If you do not specify a file name, the `mongoexport` (page 1093) writes data to standard output (e.g. `stdout`).

command line option!–forceTableScan

--forceTableScan
New in version 2.2.

Forces `mongoexport` (page 1093) to scan the data store directly: typically, `mongoexport` (page 1093) saves entries as they appear in the index of the `_id` field. Use *--forceTableScan* (page 1096) 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 1096), mongoexport (page 1093) does not use $snapshot (page 832). As a result, the export produced by mongoexport (page 1093) can reflect the state of the database at many different points in time.

> **Warning:** Use *--forceTableScan* (page 1096) with extreme caution and consideration.

Usage

In the following example, mongoexport (page 1093) exports the collection contacts from the users database from the mongod (page 1049) 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/manualopt/backups/contacts.csv. The fields.txt file contains a line-separated list of fields to export.

```
mongoexport --db users --collection contacts --csv --fieldFile fields.txt --out /opt/backups/contacts
```

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/manualsrv/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 (page 1049) connected to the data files located in the http://docs.mongodb.org/manualsrv/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 con
```

56.1.5 Diagnostic Tools

mongostat (page 1098), mongotop (page 1103), and mongosniff (page 1106) provide diagnostic information related to the current operation of a mongod (page 1049) instance.

Note: Because mongosniff (page 1106) depends on *libpcap*, most distributions of MongoDB do *not* include mongosniff (page 1106).

mongostat

Synopsis

The mongostat (page 1098) utility provides a quick overview of the status of a currently running mongod (page 1049) or mongos (page 1061) instance. mongostat (page 1098) is functionally similar to the UNIX/Linux file system utility `vmstat`, but provides data regarding `mongod` (page 1049) and `mongos` (page 1061) instances.

See also:

For more information about monitoring MongoDB, see *Monitoring for MongoDB* (page 158).

For more background on various other MongoDB status outputs see:

- *serverStatus* (page 919)
- *replSetGetStatus* (page 865)
- *dbStats* (page 904)
- *collStats* (page 900)

For an additional utility that provides MongoDB metrics see "*mongotop* (page 1103)."

`mongostat` (page 1098) connects to the `mongod` (page 1049) instance running on the local host interface on TCP port `27017`; however, `mongostat` (page 1098) can connect to any accessible remote `mongod` (page 1049) instance.

Options

mongostat

mongostat

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Returns the version of the `mongostat` (page 1098) utility.

command line option!–host <hostname><:port>

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` (page 1049) from which you want to export data. By default `mongostat` (page 1098) attempts to connect to a MongoDB instance running 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, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–port <port>

--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`* command.

command line option!–ipv6

--ipv6

Enables IPv6 support that allows mongostat (page 1098) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including mongostat (page 1098), disable IPv6 support by default.

command line option!–ssl

--ssl

New in version 2.4: MongoDB added support for SSL connections to mongod (page 1049) instances in mongostat.

Note: SSL support in mongostat is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongostat does not support connections to mongod (page 1049) instances that require client certificate validation.

Allows mongostat (page 1098) to connect to mongod (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--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* option to supply a password.

Important: This user must have sufficient credentials to run the serverStatus (page 919) command, which is the clusterAdmin (page 235) role. See *User Privilege Roles in MongoDB* (page 233) and *system.users Privilege Documents* (page 238) for more information.

---

command line option!–password <password>, -p <password>

--password <password>**, -p** <password>

Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongostat --username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, mongostat (page 1098) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>

New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, mongostat (page 1098) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See userSource (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>

New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is MONGODB-CR, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, mongostat (page 1098) also includes support for GSSAPI to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–noheaders

--noheaders

Disables the output of column or field names.

command line option!–rowcount <number>, -n <number>

--rowcount <number>, **-n** <number>

Controls the number of rows to output. Use in conjunction with the `sleeptime` argument to control the duration of a `mongostat` (page 1098) operation.

Unless *--rowcount* (page 1100) is specified, `mongostat` (page 1098) will return an infinite number of rows (e.g. value of `0`.)

command line option!–http

--http

Configures `mongostat` (page 1098) to collect data using the HTTP interface rather than a raw database connection.

command line option!–discover

--discover

With this option `mongostat` (page 1098) 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 1100) all non-*hidden members* of the replica set. When connected to a `mongos` (page 1061), `mongostat` (page 1098) will return data from all *shards* in the cluster. If a replica set provides a shard in the sharded cluster, `mongostat` (page 1098) will report on non-hidden members of that replica set.

The *mongostat --host* option is not required but potentially useful in this case.

command line option!–all

--all

Configures `mongostat` (page 1098) to return all optional *fields* (page 1100).

<sleeptime>

The final argument is the length of time, in seconds, that `mongostat` (page 1098) waits in between calls. By default `mongostat` (page 1098) returns one call every second.

`mongostat` (page 1098) returns values that reflect the operations over a 1 second period. For values of `<sleeptime>` greater than 1, `mongostat` (page 1098) averages data to reflect average operations per second.

Fields

`mongostat` (page 1098) returns values that reflect the operations over a 1 second period. When **mongostat <sleeptime>** has a value greater than 1, `mongostat` (page 1098) averages the statistics to reflect average operations per second.

`mongostat` (page 1098) 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` (page 1098) 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` (page 1098) call.

size
The amount of (virtual) memory in megabytes used by the process at the time of the last `mongostat` (page 1098) call.

res
The amount of (resident) memory in megabytes used by the process at the time of the last `mongostat` (page 1098) 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` (page 1098) will report the database that has spent the most time since the last `mongostat` (page 1098) call with a write lock.

This value represents the amount of time that the listed database spent in a locked state *combined* with the time that the `mongod` (page 1049) 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 *bytes*, received by the MongoDB instance.

This includes traffic from `mongostat` (page 1098) itself.

netOut
The amount of network traffic, in *bytes*, sent by the MongoDB instance.

This includes traffic from `mongostat` (page 1098) itself.

conn
The total number of open connections.

set
The name, if applicable, of the replica set.

repl
The replication status of the member.

Value	Replication Type
M	*master*
SEC	*secondary*
REC	recovering
UNK	unknown
SLV	*slave*

Usage

In the first example, `mongostat` (page 1098) will return data every second for 20 seconds. `mongostat` (page 1098) collects data from the `mongod` (page 1049) 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` (page 1098) returns data every 5 minutes (or 300 seconds) for as long as the program runs. `mongostat` (page 1098) collects data from the `mongod` (page 1049) 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` (page 1098) returns data every 5 minutes for an hour (12 times.) `mongostat` (page 1098) collects data from the `mongod` (page 1049) 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` will help provide a more complete snapshot of the state of an entire group of machines. If a `mongos` (page 1061) 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 (page 1103) provides a method to track the amount of time a MongoDB instance spends reading and writing data. mongotop (page 1103) provides statistics on a per-collection level. By default, mongotop (page 1103) returns values every second.

See also:

For more information about monitoring MongoDB, see *Monitoring for MongoDB* (page 158).

For additional background on various other MongoDB status outputs see:

- *serverStatus* (page 919)
- *replSetGetStatus* (page 865)
- *dbStats* (page 904)
- *collStats* (page 900)

For an additional utility that provides MongoDB metrics see "*mongostat* (page 1097)."

Options

mongotop

mongotop

command line option!–help

--help
Returns a basic help and usage text.

command line option!–verbose, -v

--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`.)

command line option!–version

--version
Print the version of the mongotop (page 1103) utility and exit.

command line option!–host <hostname><:port>

--host `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` from which you want to export data. By default mongotop (page 1103) 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`.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–port <port>

--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* command.

command line option!–ipv6

--ipv6
Enables IPv6 support that allows `mongotop` (page 1103) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongotop` (page 1103), disable IPv6 support by default.

command line option!–username <username>, -u <username>

--username <username>, **-u** <username>
Specifies a username to authenticate to the MongoDB instance, if your database requires authentication. Use in conjunction with the *mongotop* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *--username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, `mongotop` (page 1103) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, `mongotop` (page 1103) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See `userSource` (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is `MONGODB-CR`, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, `mongotop` (page 1103) also includes support for `GSSAPI` to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–locks

--locks
New in version 2.2.

Toggles the mode of `mongotop` (page 1103) to report on use of per-database *locks* (page 920). These data are useful for measuring concurrent operations and lock percentage.

<sleeptime>
The final argument is the length of time, in seconds, that `mongotop` (page 1103) waits in between calls. By default `mongotop` (page 1103) returns data every second.

Fields

mongotop (page 1103) returns time values specified in milliseconds (ms.)

mongotop (page 1103) only reports active namespaces or databases, depending on the *--locks* (page 1104) 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 (page 1066) shell to generate activity to affect the output of mongotop (page 1103).

`mongotop.`**`ns`**
Contains the database namespace, which combines the database name and collection.

Changed in version 2.2: If you use the *--locks* (page 1104), the ns (page 1105) field does not appear in the mongotop (page 1103) output.

`mongotop.`**`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 (page 1103) with the *--locks* (page 1104) option.

`mongotop.`**`total`**
Provides the total amount of time that this mongod (page 1049) spent operating on this namespace.

`mongotop.`**`read`**
Provides the amount of time that this mongod (page 1049) spent performing read operations on this namespace.

`mongotop.`**`write`**
Provides the amount of time that this mongod (page 1049) spent performing write operations on this namespace.

`mongotop.`**<timestamp>**
Provides a time stamp for the returned data.

Use

By default mongotop (page 1103) connects to the MongoDB instance running on the localhost port `27017`. However, mongotop (page 1103) can optionally connect to remote mongod (page 1049) instances. See the *mongotop options* (page 1103) for more information.

To force mongotop (page 1103) to return less frequently specify a number, in seconds at the end of the command. In this example, mongotop (page 1103) 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 a mongotop (page 1103) report every 5 minutes, use the following command:

```
mongotop 300
```

To report the use of per-database locks, use *mongotop --locks*, 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 (page 1106) provides a low-level operation tracing/sniffing view into database activity in real time. Think of mongosniff (page 1106) as a MongoDB-specific analogue of tcpdump for TCP/IP network traffic. Typically, mongosniff (page 1106) is most frequently used in driver development.

Note: mongosniff (page 1106) 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 (page 1106) 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 alternative to mongosniff (page 1106), Wireshark, a popular network sniffing tool is capable of inspecting and parsing the MongoDB wire protocol.

Options

mongosniff

mongosniff

command line option!–help

--help
Returns a basic help and usage text.

command line option!–forward <host><:port>

--forward <host><:port>
Declares a host to forward all parsed requests that the mongosniff (page 1106) intercepts to another mongod (page 1049) instance and issue those operations on that database instance.

Specify the target host name and port in the <host><:port> format.

To connect to a replica set, you can specify the replica set seed name, and a seed list of set members, in the following format:

```
<replica_set_name>/<hostname1><:port>,<hostname2:<port>,...
```

command line option!–source <NET [interface]>, <FILE [filename]>, <DIAGLOG [filename]>

--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* option.

command line option!–objcheck

--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` (page 1106).

<port>
Specifies alternate ports to sniff for traffic. By default, `mongosniff` (page 1106) watches for MongoDB traffic on port 27017. Append multiple port numbers to the end of `mongosniff` (page 1106) to monitor traffic on multiple ports.

Usage

Use the following command to connect to a `mongod` (page 1049) or `mongos` (page 1061) 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` (page 1049) or `mongos` (page 1061) running on the localhost interface and port 27018, for driver development and troubleshooting:

```
mongosniff --objcheck --source NET lo 27018
```

Build `mongosniff`

To build `mongosniff` yourself, Linux users can use the following procedure:

1. Obtain prerequisites using your operating systems package management software. Dependencies include:
    - `libpcap` - to capture network packets.
    - `git` - to download the MongoDB source code.
    - `scons` and a C++ compiler - to build `mongosniff` (page 1106).
2. Download a copy of the MongoDB source code using `git`:

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

3. Issue the following sequence of commands to change to the `mongo/` directory and build `mongosniff` (page 1106):

    ```
    cd mongo
    scons mongosniff
    ```

Note: If you run `scons mongosniff` before installing `libpcap` you must run `scons clean` before you can build `mongosniff` (page 1106).

`mongoperf`

Synopsis

`mongoperf` (page 1108) is a utility to check disk I/O performance independently of MongoDB.

It times tests of random disk I/O and presents the results. You can use `mongoperf` (page 1108) for any case apart from MongoDB. The `mmf` (page 1109) `true` mode is completely generic. In that mode it is somewhat analogous to tools such as bonnie++ (albeit mongoperf is simpler).

Specify options to `mongoperf` (page 1108) using a JavaScript document.

See also:

- bonnie
- bonnie++
- Output from an example run
- Checking Disk Performance with the mongoperf Utility

Options

`mongoperf`

`mongoperf`

command line option!–help

`--help`

Displays the options to `mongoperf` (page 1108). Specify options to `mongoperf` (page 1108) with a JSON document described in the *Configuration Fields* (page 1109) section.

`<jsonconfig>`

`mongoperf` (page 1108) accepts configuration options in the form of a file that holds a *JSON* document. You must stream the content of this file into `mongoperf` (page 1108), as in the following operation:

```
mongoperf < config
```

In this example `config` is the name of a file that holds a JSON document that resembles the following example:

```
{
  nThreads:<n>,
  fileSizeMB:<n>,
  sleepMicros:<n>,
  mmf:<bool>,
  r:<bool>,
  w:<bool>,
  recSizeKB:<n>,
  syncDelay:<n>
}
```

See the *Configuration Fields* (page 1109) section for documentation of each of these fields.

Configuration Fields

`mongoperf.`**`nThreads`**

Type: Integer.

Default: 1

Defines the number of threads `mongoperf` (page 1108) will use in the test. To saturate your system's storage system you will need multiple threads. Consider setting `nThreads` (page 1109) to `16`.

`mongoperf.`**`fileSizeMB`**

Type: Integer.

Default: 1 megabyte (i.e. 1024^2 bytes)

Test file size.

`mongoperf.`**`sleepMicros`**

Type: Integer.

Default: 0

`mongoperf` (page 1108) will pause for the number of specified `sleepMicros` (page 1109) divided by the `nThreads` (page 1109) between each operation.

`mongoperf.`**`mmf`**

Type: Boolean.

Default: `false`

Set `mmf` (page 1109) to `true` to use memory mapped files for the tests.

Generally:

- when `mmf` (page 1109) is `false`, `mongoperf` (page 1108) tests direct, physical, I/O, without caching. Use a large file size to test heavy random I/O load and to avoid I/O coalescing.
- when `mmf` (page 1109) is `true`, `mongoperf` (page 1108) runs tests of the caching system, and can use normal file system cache. Use `mmf` (page 1109) in this mode to test file system cache behavior with memory mapped files.

`mongoperf.`**`r`**

Type: Boolean.

Default: `false`

Set `r` (page 1109) to `true` to perform reads as part of the tests.

Either `r` (page 1109) or `w` (page 1109) must be `true`.

`mongoperf.`**`w`**

Type: Boolean.

Default: `false`

Set `w` (page 1109) to `true` to perform writes as part of the tests.

Either `r` (page 1109) or `w` (page 1109) must be `true`.

`mongoperf.`**`recSizeKB`**

New in version 2.4.

Type: Integer.

Default: 4 kb

The size of each write operation.

mongoperf.**syncDelay**

Type: Integer.

Default: 0

Seconds between disk flushes. mongoperf.syncDelay (page 1110) is similar to *--syncdelay* for mongod (page 1049).

The syncDelay (page 1110) controls how frequently mongoperf (page 1108) performs an asynchronous disk flush of the memory mapped file used for testing. By default, mongod (page 1049) performs this operation every 60 seconds. Use syncDelay (page 1110) to test basic system performance of this type of operation.

Only use syncDelay (page 1110) in conjunction with mmf (page 1109) set to `true`.

The default value of `0` disables this.

Use

```
mongoperf < jsonconfigfile
```

Replace `jsonconfigfile` with the path to the mongoperf (page 1108) configuration. You may also invoke mongoperf (page 1108) in the following form:

```
echo "{nThreads:16,fileSizeMB:1000,r:true}" | ./mongoperf
```

In this operation:

- mongoperf (page 1108) tests direct physical random read io's, using 16 concurrent reader threads.
- mongoperf (page 1108) uses a 1 gigabyte test file.

Consider using `iostat`, as invoked in the following example to monitor I/O performance during the test.

```
iostat -xm 2
```

56.1.6 GridFS

mongofiles (page 1110) provides a command-line interact to a MongoDB *GridFS* storage system.

mongofiles

mongofiles

Synopsis

The mongofiles (page 1110) 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 (page 1110) commands have the following form:

```
mongofiles <options> <commands> <filename>
```

The components of the mongofiles (page 1110) command are:

1. *Options* (page 1111). You may use one or more of these options to control the behavior of mongofiles (page 1110).

2. *Commands* (page 1113). Use one of these commands to determine the action of `mongofiles` (page 1110).
3. A filename which is either: the name of a file on your local's file system, or a GridFS object.

`mongofiles` (page 1110), like `mongodump` (page 1075), `mongoexport` (page 1093), `mongoimport` (page 1089), and `mongorestore` (page 1079), can access data stored in a MongoDB data directory without requiring a running `mongod` (page 1049) instance, if no other `mongod` (page 1049) is running.

Important: For *replica sets*, `mongofiles` (page 1110) can only read from the set's '*primary*.

Options

`mongofiles`

command line option!–help

`--help`
Returns a basic help and usage text.

command line option!–verbose, -v

`--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`.)

command line option!–version

`--version`
Returns the version of the `mongofiles` (page 1110) utility.

command line option!–host <hostname><:port>

`--host` `<hostname><:port>`
Specifies a resolvable hostname for the `mongod` (page 1049) that holds your GridFS system. By default `mongofiles` (page 1110) 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.

command line option!–port <port>

`--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`* command.

command line option!–ipv6

`--ipv6`
Enables IPv6 support that allows `mongofiles` (page 1110) to connect to the MongoDB instance using an IPv6 network. All MongoDB programs and processes, including `mongofiles` (page 1110), disable IPv6 support by default.

command line option!–ssl

`--ssl`
New in version 2.4: MongoDB added support for SSL connections to `mongod` (page 1049) instances in mongofiles.

Note: SSL support in mongofiles is not compiled into the default distribution of MongoDB. See *Connect to MongoDB with SSL* (page 179) for more information on SSL and MongoDB.

Additionally, mongofiles does not support connections to `mongod` (page 1049) instances that require client certificate validation.

Allows `mongofiles` (page 1110) to connect to `mongod` (page 1049) instance over an SSL connection.

command line option!–username <username>, -u <username>

--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* option to supply a password.

command line option!–password <password>, -p <password>

--password <password>, **-p** <password>
Specifies a password to authenticate to the MongoDB instance. Use in conjunction with the *mongofiles --username* option to supply a username.

If you specify a *--username* without the *--password* (page 1067) option, `mongofiles` (page 1110) will prompt for a password interactively.

command line option!–authenticationDatabase <dbname>

--authenticationDatabase <dbname>
New in version 2.4.

Specifies the database that holds the user's (e.g *--username*) credentials.

By default, `mongofiles` (page 1110) assumes that the database specified to the *--db* (page 1081) argument holds the user's credentials, unless you specify *--authenticationDatabase* (page 1067).

See `userSource` (page 239), *system.users Privilege Documents* (page 238) and *User Privilege Roles in MongoDB* (page 233) for more information about delegated authentication in MongoDB.

command line option!–authenticationMechanism <name>

--authenticationMechanism <name>
New in version 2.4.

Specifies the authentication mechanism. By default, the authentication mechanism is `MONGODB-CR`, which is the MongoDB challenge/response authentication mechanism. In MongoDB Enterprise, `mongofiles` (page 1110) also includes support for `GSSAPI` to handle Kerberos authentication.

See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information about Kerberos authentication.

command line option!–dbpath <path>

--dbpath <path>
Specifies the directory of the MongoDB data files. If used, the *--dbpath* (page 1081) option enables `mongofiles` (page 1110) to attach directly to local data files interact with the GridFS data without the `mongod` (page 1049). To run with *--dbpath* (page 1081), `mongofiles` (page 1110) needs to lock access to the data directory: as a result, no `mongod` (page 1049) can access the same path while the process runs.

command line option!–directoryperdb

--directoryperdb
Use the *--directoryperdb* (page 1081) in conjunction with the corresponding option to `mongod` (page 1049), which allows `mongofiles` (page 1110) when running with the *--dbpath* (page 1081) 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 1081) option.

command line option!–journal

--journal
Allows mongofiles (page 1110) operations to use the durability *journal* when running with *--dbpath* (page 1081) to ensure that the database maintains a recoverable state. This forces mongofiles (page 1110) to record all data on disk regularly.

command line option!–db <db>, -d <db>

--db <db>, **-d** <db>
Use the *--db* (page 1081) option to specify the MongoDB database that stores or will store the GridFS files.

command line option!–collection <collection>, -c <collection>

--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.

command line option!–local <filename>, -l <filename>

--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 (page 1110) assumes that this reflects the file's name on the local file system. This setting overrides this default.

command line option!–type <MIME>, t <MIME>

--type <MIME>, t <MIME>
Provides the ability to specify a *MIME* type to describe the file inserted into GridFS storage. mongofiles (page 1110) omits this option in the default operation.

Use only with **mongofiles put** operations.

command line option!–replace, -r

--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.

Commands

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 (page 1110) assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* 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 (page 1110) assumes that this reflects the name the file has on the local file system. If the local filename is different use the *mongofiles --local* option.

delete <filename>
Delete the specified file from GridFS storage.

Examples

To return a list of all files in a *GridFS* collection in the records database, use the following invocation at the system shell:

```
mongofiles -d records list
```

This mongofiles (page 1110) instance will connect to the mongod (page 1049) instance running on the 27017 localhost interface to specify the same operation on a different port or hostname, and issue a command that resembles one of the following:

```
mongofiles --port 37017 -d records list
mongofiles --hostname db1.example.net -d records list
mongofiles --hostname db1.example.net --port 37017 -d records list
```

Modify any of the following commands as needed if you're connecting the mongod (page 1049) instances on different ports or hosts.

To upload a file named 32-corinth.lp to the GridFS collection in the records database, you can use the following command:

```
mongofiles -d records put 32-corinth.lp
```

To delete the 32-corinth.lp file from this GridFS collection in the records database, you can use the following command:

```
mongofiles -d records delete 32-corinth.lp
```

To search for files in the GridFS collection in the records database that have the string corinth in their names, you can use following command:

```
mongofiles -d records search corinth
```

To list all files in the GridFS collection in the records database that begin with the string 32, you can use the following command:

```
mongofiles -d records list 32
```

To fetch the file from the GridFS collection in the records database named 32-corinth.lp, you can use the following command:

```
mongofiles -d records get 32-corinth.lp
```

56.1.7 Run Time Configuration

Configuration File Options

Synopsis

Administrators and users can control `mongod` (page 1049) or `mongos` (page 1061) instances at runtime either directly from *mongod's command line arguments* (page 1049) 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` (page 1049) or `mongos` (page 1061) 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.

Settings

verbose

Default: false

Increases the amount of internal reporting returned on standard output or in the log file generated by `logpath` (page 1116). To enable `verbose` (page 1115) or to enable increased verbosity with `vvvv` (page 1115), set these options as in the following example:

```
verbose = true
vvvv = true
```

MongoDB has the following levels of verbosity:

v

Default: false

Alternate form of `verbose` (page 1115).

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.

port
Default: 27017

Specifies a TCP port for the `mongod` (page 1049) or `mongos` (page 1061) instance to listen for client connections. UNIX-like systems require root access for ports with numbers lower than 1024.

bind_ip
Default: All interfaces.

Set this option to configure the `mongod` (page 1049) or `mongos` (page 1061) process to bind to and listen for connections from applications on this address. You may attach `mongod` (page 1049) or `mongos` (page 1061) 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` (page 1049) 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` (page 1049) or `mongos` (page 1061) 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 1061) 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 1116), 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 1116) to a value higher than *20000*.

objcheck
Default: true

Changed in version 2.4: The default setting for `objcheck` (page 1116) became `true` in 2.4. In earlier versions `objcheck` (page 1116) was `false` by default.

Forces the `mongod` (page 1049) to validate all requests from clients upon receipt to ensure that clients never insert invalid documents into the database. For objects with a high degree of sub-document nesting, `objcheck` (page 1116) can have a small impact on performance. You can set `noobjcheck` (page 1116) to disable object checking at run-time.

noobjcheck
New in version 2.4.

Default: false

Disables the default object validation that MongoDB performs on all incoming BSON documents.

logpath
Default: None. (i.e. `http://docs.mongodb.org/manualdev/stdout`)

Specify the path to a file name for the log file that will hold all diagnostic logging information.

Unless specified, `mongod` (page 1049) will output all log information to the standard output. Unless `logappend` (page 1117) 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 1117) 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

New in version 2.1.0.

Sends all logging output to the host's *syslog* system rather than to standard output or a log file as with `logpath` (page 1116).

Warning: You cannot use `syslog` (page 1117) with `logpath` (page 1116).

pidfilepath

Default: None.

Specify a file location to hold the "*PID*" or process ID of the `mongod` (page 1049) process. Useful for tracking the `mongod` (page 1049) process in combination with the `fork` (page 1117) setting.

Without a specified `pidfilepath` (page 1117), `mongos` (page 1061) creates no PID file.

Without this option, `mongod` (page 1049) 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 212)"

nounixsocket

Default: false

Set to `true` to disable listening on the UNIX socket.

MongoDB always creates and listens on the UNIX socket, unless `nounixsocket` (page 1117) is set, or `bind_ip` (page 1116) is *not* set, or `bind_ip` (page 1116) does *not* specify `127.0.0.1`.

unixSocketPrefix

Default: `http://docs.mongodb.org/manualtmp`

Specifies a path for the UNIX socket. Unless this option has a value `mongod` (page 1049) creates a socket with `http://docs.mongodb.org/manualtmp` as a prefix.

MongoDB will *always* create and listen on a UNIX socket, unless `nounixsocket` (page 1117) is set, `bind_ip` (page 1116) is *not* set. or `bind_ip` (page 1116) does *not* specify `127.0.0.1`.

fork

Default: false

Set to `true` to enable a *daemon* mode for `mongod` (page 1049) 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 1066). 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` (page 1049) 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 1116) option.

dbpath

Default: `http://docs.mongodb.org/manualdata/db/`

Set this value to designate a directory for the `mongod` (page 1049) instance to store its data. Typical locations include: `http://docs.mongodb.org/manualsrv/mongodb`, `http://docs.mongodb.org/manualvar/lib/mongodb` or `http://docs.mongodb.org/manualopt/mongo`

Unless specified, `mongod` (page 1049) will look for data files in the default `http://docs.mongodb.org/manualdata/db` directory. (Windows systems use the `\data\db` directory.) If you installed using a package management system. Check the `/etc/mongodb.conf` file provided by your packages to see the configuration of the `dbpath` (page 1118).

diaglog

Default: 0

Creates a very verbose *diagnostic log* for troubleshooting and recording various errors. MongoDB writes these log files in the `dbpath` (page 1118) directory and names them `diaglog.<time in hex>`, where `<time-in-hex>` is the initiation time of logging as a hexadecimal 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` (page 1106) tool to replay this output for investigation. Given a typical diaglog file, located at `http://docs.mongodb.org/manualdata/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 1118) is for internal use and not intended for most users.

> **Warning:** Setting the diagnostic level to `0` will cause `mongod` (page 1049) to stop writing data to the *diagnostic log* file. However, the `mongod` (page 1049) 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` (page 1049) 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 1118) 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.

Warning: If you have an existing `mongod` (page 1049) instance and `dbpath` (page 1118), and you want to enable `directoryperdb` (page 1118), you **must** migrate your existing databases to directories before setting `directoryperdb` (page 1118) to access those databases.

Example
Given a `dbpath` (page 1118) directory with the following items:

```
journal
mongod.lock
local.0
local.1
local.ns
test.0
test.1
test.ns
```

To enable `directoryperdb` (page 1118) you would need to modify the `dbpath` (page 1118) to resemble the following:

```
journal
mongod.lock
local/local.0
local/local.1
local/local.ns
test/test.0
test/test.1
test/test.ns
```

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 1119) enabled, and set `smallfiles` (page 1122) to true to reduce the size of the data and journal files.

Note: You must use `nojournal` (page 1120) to disable journaling on 64-bit systems.

journalCommitInterval
Default: 100 or 30

Set this value to specify the maximum amount of time for `mongod` (page 1049) to allow between journal operations. Lower values increase the durability of the journal, at the possible expense of disk performance.

The default journal commit interval is 100 milliseconds if a single block device (e.g. physical volume, RAID device, or LVM volume) contains both the journal and the data files.

If different block devices provide the journal and data files the default journal commit interval is 30 milliseconds.

This option accepts values between 2 and 300 milliseconds.

To force mongod (page 1049) to commit to the journal more frequently, you can specify `j:true`. When a write operation with `j:true` is pending, mongod (page 1049) will reduce journalCommitInterval (page 1119) to a third of the set value.

ipv6

Default: false

Set to `true` to IPv6 support to allow clients to connect to mongod (page 1049) using IPv6 networks. mongod (page 1049) disables IPv6 support by default in mongod (page 1049) 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 1118) option.

nohttpinterface

Default: false

Set to `true` to disable the HTTP interface. This command will override the rest (page 1121) and disable the HTTP interface if you specify both.

Changed in version 2.1.2: The nohttpinterface (page 1120) option is not available for mongos (page 1061) 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 (page 1049) enables journaling in 64-bit versions after v2.0.

Note: You must use journal (page 1119) to enable journaling on 32-bit systems.

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. The maximum size is 2047 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.

See `Limits on namespaces` (page 1139).

profile
Default: 0

Modify this value to changes the level of database profiling, which inserts information about operation performance into output of `mongod` (page 1049) or the log file if specified by `logpath` (page 1116). 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` (page 1049) 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 1121) setting.

quotaFiles
Default: 8

Modify limit on the number of data files per database. This option requires the `quota` (page 1121) 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 129) or in a *control script*.

Use the *mongod --repair* option to access this functionality.

Note: Because `mongod` (page 1049) rewrites all of the database files during the repair routine, if you do not run `repair` (page 1121) under the same user account as `mongod` (page 1049) usually runs, you will need to run `chown` on your database files to correct the permissions before starting `mongod` (page 1049) again.

repairpath
Default: A `_tmp` directory in the `dbpath` (page 1118).

Specify the path to the directory containing MongoDB data files, to use in conjunction with the `repair` (page 1121) setting or *mongod --repair* operation. Defaults to a `_tmp` directory within the `dbpath` (page 1118).

slowms
Default: 100

Specify values in milliseconds.

Sets the threshold for `mongod` (page 1049) 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` (page 1049) the profiler writes to the `system.profile` collection.

See also:

"`profile` (page 1121)"

smallfiles

Default: false

Set to `true` to modify MongoDB to use a smaller default data file size. Specifically, `smallfiles` (page 1122) reduces the initial size for data files and limits them to 512 megabytes. The `smallfiles` (page 1122) setting also reduces the size of each *journal* files from 1 gigabyte to 128 megabytes.

Use the `smallfiles` (page 1122) setting if you have a large number of databases that each hold a small quantity of data. The `smallfiles` (page 1122) setting can lead `mongod` (page 1049) to create many files, which may affect performance for larger databases.

syncdelay

Default: 60

`mongod` (page 1049) writes data very quickly to the journal, and lazily to the data files. `syncdelay` (page 1122) controls how much time can pass before MongoDB flushes data to the *database files* via an *fsync* operation. The default setting is 60 seconds. In almost every situation you should not set this value and use the default setting.

The `serverStatus` (page 919) command reports the background flush thread's status via the `backgroundFlushing` (page 926) field.

`syncdelay` (page 1122) has no effect on the `journal` (page 1119) files or *journaling* (page 155).

> **Warning:** If you set `syncdelay` (page 1122) to `0`, MongoDB will not sync the memory mapped files to disk. Do not set this value on production systems.

sysinfo

Default: false

When set to `true`, `mongod` (page 1049) 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`* command. When running with the `sysinfo` (page 1122), only `mongod` (page 1049) 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 1118) to the latest version, if needed.

This option only affects the operation of `mongod` (page 1049) if the data files are in an old format.

When specified for a `mongos` (page 1061) 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.

quiet

Default: false

Runs the mongod (page 1049) or mongos (page 1061) instance in a quiet mode that attempts to limit the amount of output. This option suppresses:

- output from *database commands*, including drop (page 884), dropIndexes (page 888), diagLogging (page 906), validate (page 908), and clean (page 890).
- replication activity.
- connection accepted events.
- connection closed events.

Note: For production systems this option is **not** recommended as it may make tracking problems during particular connections much more difficult.

setParameter

New in version 2.4.

Specifies an option to configure on startup. Specify multiple options with multiple setParameter (page 1123) options. See *mongod Parameters* (page 1129) for full documentation of these parameters. The setParameter (page 894) database command provides access to many of these parameters.

Declare all setParameter (page 894) settings in this file using the following form:

```
setParameter = <parameter>=<value>
```

For mongod (page 1049) the following options are available using setParameter (page 1123):

- enableLocalhostAuthBypass (page 1129)
- enableTestCommands (page 1129)
- journalCommitInterval (page 1129)
- logLevel (page 1130)
- logUserIds (page 1130)
- notablescan (page 1130)
- quiet (page 1131)
- replApplyBatchSize (page 1130)
- replIndexPrefetch (page 1130)
- supportCompatibilityFormPrivilegeDocuments (page 1131)
- syncdelay (page 1131)
- textSearchEnabled (page 1132)
- traceExceptions (page 1131)

For mongos (page 1061) the following options are available using setParameter (page 1123):

- enableLocalhostAuthBypass (page 1129)
- enableTestCommands (page 1129)

- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `supportCompatibilityFormPrivilegeDocuments` (page 1131)
- `syncdelay` (page 1131)
- `textSearchEnabled` (page 1132)

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 367)," "*Replica Set Deployment Tutorials* (page 416)," and "*Replica Set Configuration* (page 473)"

oplogSize

Specifies a maximum size in megabytes for the replication operation log (e.g. *oplog*.) `mongod` (page 1049) 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` (page 1049) has created the oplog for the first time, changing `oplogSize` (page 1124) 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 member with a snapshot of the *dbpath* of another member of the set. Otherwise the `mongod` (page 1049) will attempt to perform an initial sync, as though the member were a new member.

> **Warning:** If the data is not perfectly synchronized *and* `mongod` (page 1049) starts with `fastsync` (page 1124), 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 can only use `replIndexPrefetch` (page 1124) in conjunction with `replSet` (page 1124).

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` (page 1049) 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 1125) setting to specify the *master* instance from which this *slave* instance will replicate

only
Default: <>

Used with the `slave` (page 1125) option, `only` (page 1125) specifies only a single *database* to replicate.

slaveDelay
Default: 0

Used with the `slave` (page 1125) setting, `slaveDelay` (page 1125) 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 1125) setting, set `autoresync` (page 1125) to `true` to force the *slave* to automatically resync if it is more than 10 seconds behind the master. This setting may be problematic if the `oplogSize` (page 1124) of the *oplog* is too small. If the *oplog* is 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 1125) option to `false`, the slave will not attempt an automatic resync more than once in a ten minute period.

Sharded Cluster Options

configsvr
Default: false

Set this value to `true` to configure this `mongod` (page 1049) 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 a `mongod` (page 1049) with this option is `27019` and the default `dbpath` (page 1118) directory is `http://docs.mongodb.org/manualdata/configdb`, unless specified.

Changed in version 2.2: `configsvr` (page 1125) also sets `smallfiles` (page 1122).

Changed in version 2.4: `configsvr` (page 1125) creates a local *oplog*.

Do not use `configsvr` (page 1125) with `replSet` (page 1124) or `shardsvr` (page 1125). Config servers cannot be a shard server or part of a *replica set*.

default port for `mongod` (page 1049) with this option is `27019` and `mongod` (page 1049) writes all data files to the `http://docs.mongodb.org/manualconfigdb` sub-directory of the `dbpath` (page 1118) directory.

shardsvr
Default: false

Set this value to `true` to configure this `mongod` (page 1049) instance as a shard in a partitioned cluster. The default port for these instances is `27018`. The only affect of `shardsvr` (page 1125) is to change the port number.

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 1061) processes.

Note: `mongos` (page 1061) instances read from the first *config server* in the list provided. All `mongos` (page 1061) instances **must** specify the hosts to the `configdb` (page 1126) setting in the same order.

If your configuration databases reside in more that one data center, order the hosts in the `configdb` (page 1126) setting so that the config database that is closest to the majority of your `mongos` (page 1061) instances is first servers in the list.

Warning: Never remove a config server from the `configdb` (page 1126) 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 1061) instance.

This setting only affects `mongos` (page 1061) 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 1061) processes. Furthermore, `chunkSize` (page 1126) *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 538)" procedure if you need to change the chunk size on an existing sharded cluster.

localThreshold

New in version 2.2.

`localThreshold` (page 1126) 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 1126) in milliseconds. The default value is `15`, which corresponds to the default value in all of the client *drivers* (page 575).

This setting only affects `mongos` (page 1061) processes.

When `mongos` (page 1061) receives a request that permits reads to *secondary* members, the `mongos` (page 1061) 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 1126), `mongos` (page 1061) will construct the list of replica members that are within the latency allowed by this value.

- The `mongos` (page 1061) 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 1126) 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 1061) recalculates the average.

See the *Member Selection* (page 403) section of the *read preference* (page 398) documentation for more information.

noAutoSplit

`noAutoSplit` (page 1127) is for internal use and is only available on `mongos` (page 1061) instances.

New in version 2.0.7.

`noAutoSplit` (page 1127) prevents `mongos` (page 1061) from automatically inserting metadata splits in a *sharded collection*. If set on all `mongos` (page 1061), this will prevent MongoDB from creating new chunks as the data in a collection grows.

Because any `mongos` (page 1061) in a cluster can create a split, to totally disable splitting in a cluster you must set `noAutoSplit` (page 1127) on all `mongos` (page 1061).

> **Warning:** With `noAutoSplit` (page 1127) enabled, the data in your sharded cluster may become imbalanced over time. Enable with caution.

SSL Options

sslOnNormalPorts

New in version 2.2.

> **Note:** The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Enables SSL for `mongod` (page 1049) or `mongos` (page 1061). With `sslOnNormalPorts` (page 1127), a `mongod` (page 1049) or `mongos` (page 1061) requires SSL encryption for all connections on the default MongoDB port, or the port specified by `port` (page 1116). By default, `sslOnNormalPorts` (page 1127) is disabled.

sslPEMKeyFile

New in version 2.2.

> **Note:** The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains both the SSL certificate and key. Specify the file name of the `.pem` file using relative or absolute paths

When using `sslOnNormalPorts` (page 1127), you must specify `sslPEMKeyFile` (page 1127).

sslPEMKeyPassword

New in version 2.2.

> **Note:** The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the password to de-crypt the certificate-key file (i.e. `sslPEMKeyFile` (page 1127)). Only use `sslPEMKeyPassword` (page 1127) if the certificate-key file is encrypted. In all cases, `mongod` (page 1049) or `mongos` (page 1061) will redact the password from all logging and reporting output.

Changed in version 2.4: `sslPEMKeyPassword` (page 1127) is only needed when the private key is encrypted. In earlier versions `mongod` (page 1049) or `mongos` (page 1061) would require `sslPEMKeyPassword` (page 1127) whenever using `sslOnNormalPorts` (page 1127), even when the private key was not encrypted.

sslCAFile

New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains the root certificate chain from the Certificate Authority. Specify the file name of the `.pem` file using relative or absolute paths

sslCRLFile

New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Specifies the `.pem` file that contains the Certificate Revocation List. Specify the file name of the `.pem` file using relative or absolute paths

sslWeakCertificateValidation

New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

Disables the requirement for SSL certificate validation, that `sslCAFile` (page 1128) enables. With `sslWeakCertificateValidation` (page 1128), `mongod` (page 1049) or `mongos` (page 1061) will accept connections if the client does not present a certificate when establishing the connection.

If the client presents a certificate and `mongod` (page 1049) or `mongos` (page 1061) has `sslWeakCertificateValidation` (page 1128) enabled, `mongod` (page 1049) or `mongos` (page 1061) will validate the certificate using the root certificate chain specified by `sslCAFile` (page 1128), and reject clients with invalid certificates.

Use `sslWeakCertificateValidation` (page 1128) if you have a mixed deployment that includes clients that do not or cannot present certificates to `mongod` (page 1049) or `mongos` (page 1061).

sslFIPSMode

New in version 2.4.

Note: The default distribution of MongoDB does **not** contain support for SSL. To use SSL you can either compile MongoDB with SSL support or use MongoDB Enterprise. See *Connect to MongoDB with SSL* (page 179) for more information about SSL and MongoDB.

When specified, `mongod` (page 1049) or `mongos` (page 1061) will use the FIPS mode of the installed OpenSSL library. Your system must have a FIPS compliant OpenSSL library to use `sslFIPSMode` (page 1128).

`mongod` Parameters

Changed in version 2.4.

Synopsis

MongoDB provides a number of configuration options that are accessible via the *--setParameter* option to `mongod` (page 1049). This document documents all of these options.

For additional run time configuration options, see *Configuration File Options* (page 1115) and *Manual Page for mongod* (page 1049).

Parameters

enableLocalhostAuthBypass

New in version 2.4.

Specify `0` to disable localhost authentication bypass. Enabled by default.

`enableLocalhostAuthBypass` (page 1129) is not available using `setParameter` (page 894) database command. Use the `setParameter` (page 1123) option in the configuration file or the *--setParameter* option on the command line.

enableTestCommands

New in version 2.4.

`enableTestCommands` (page 1129) enables a set of internal commands useful for internal testing operations. `enableTestCommands` (page 1129) is only available when starting `mongod` (page 1049) and you cannot use `setParameter` (page 894) to modify this parameter. Consider the following `mongod` (page 1049) innovation, which sets `enableTestCommands` (page 1129):

```
mongod --setParameter enableTestCommands=1
```

`enableTestCommands` (page 1129) provides access to the following internal commands:

- `captrunc` (page 940)
- `configureFailPoint` (page 943)
- `emptycapped` (page 940)
- `godinsert` (page 941)
- `_hashBSONElement` (page 941)
- `journalLatencyTest` (page 942)
- `replSetTest` (page 943)
- `_skewClockCommand` (page 943)
- `sleep` (page 942)
- `_testDistLockWithSkew` (page 939)
- `_testDistLockWithSyncCluster` (page 939)

journalCommitInterval

Specify an integer between `1` and `500` signifying the number of milliseconds (ms) between journal commits.

Consider the following example which sets the `journalCommitInterval` (page 1129) to `200` ms:

```
use admin
db.runCommand( { setParameter: 1, journalCommitInterval: 200 } )
```

See also:

`journalCommitInterval` (page 1119).

`logUserIds`

New in version 2.4.

Specify `1` to enable logging of userids.

Disabled by default.

`logLevel`

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` (page 1130) to `2`:

```
use admin
db.runCommand( { setParameter: 1, logLevel: 2 } )
```

See also:

`verbose` (page 1115).

`notablescan`

Specify whether queries must use indexes. If `1`, queries that perform a table scan instead of using an index will fail.

Consider the following example which sets `notablescan` (page 1130) to true:

```
use admin
db.runCommand( { setParameter: 1, notablescan: 1 } )
```

See also:

`notablescan` (page 1120)

`replIndexPrefetch`

New in version 2.2.

Use `replIndexPrefetch` (page 1130) in conjunction with `replSet` (page 1124). 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` (page 1049) from loading *any* index into memory.

`replApplyBatchSize`

New in version 2.4.

Specify the number of oplog entries to apply as a single batch. `replApplyBatchSize` (page 1130) must be an integer between 1 and 1024. This option only applies to replica set members when they are in the *secondary* state.

Batch sizes must be `1` for members with `slaveDelay` (page 1125) configured.

saslHostName

New in version 2.4.

saslHostName (page 1130) overrides MongoDB's default hostname detection for the purpose of configuring SASL and Kerberos authentication.

saslHostName (page 1130) does not affect the hostname of the mongod (page 1049) or mongos (page 1061) instance for any purpose beyond the configuration of SASL and Kerberos.

You can only set saslHostName (page 1130) during start-up, and cannot change this setting using the setParameter (page 894) database command.

Note: saslHostName (page 1130) supports Kerberos authentication and is only included in MongoDB Enterprise. See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information.

supportCompatibilityFormPrivilegeDocuments

New in version 2.4.

supportCompatibilityFormPrivilegeDocuments (page 1131) is not available using setParameter (page 894) database command. Use the setParameter (page 1123) option in the configuration file or the `--setParameter` option on the command line.

syncdelay

Specify the interval in seconds between *fsync* operations where mongod (page 1049) flushes its working memory to disk. By default, mongod (page 1049) flushes memory to disk every 60 seconds. In almost every situation you should not set this value and use the default setting.

Consider the following example which sets the `syncdelay` to `60` seconds:

```
db = db.getSiblingDB("admin")
db.runCommand( { setParameter: 1, syncdelay: 60 } )
```

See also:

syncdelay (page 1122) and journalCommitInterval (page 1129).

traceExceptions

New in version 2.2.

Configures mongod (page 1049) log full stack traces on assertions or errors. If `1`, mongod (page 1049) 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 1122)

quiet

Sets quiet logging mode. If `1`, mongod (page 1049) will go into a quiet logging mode which will not log the following events/activities:

- connection events;
- the drop (page 884) command, the dropIndexes (page 888) command, the diagLogging (page 906) command, the validate (page 908) command, and the clean (page 890) command; and
- replication synchronization activities.

Consider the following example which sets the `quiet` to `1`:

```
db = db.getSiblingDB("admin")
db.runCommand( { setParameter: 1, quiet: 1 } )
```

See also:

`quiet` (page 1123)

`textSearchEnabled`

New in version 2.4.

> **Warning:**
> - Do **not** enable or use text search on production systems.
> - Text indexes have significant storage requirements and performance costs. See *text Indexes* (page 318) for more information.

Enables the *text search* (page 353) feature. You must enable the feature before creating or accessing a text index.

```
mongod --setParameter textSearchEnabled=true
```

If the flag is not enabled, you cannot create *new* `text` indexes, and you cannot perform text searches. However, MongoDB will continue to maintain existing `text` indexes.

releaseConnectionsAfterResponse>

New in version 2.2.4: and 2.4.2

Changes the behavior of connection pool that `mongos` (page 1061) uses to connect to the shards. As a result, each `mongos` (page 1061) should need to maintain fewer connections to each shard. When enabled, the `mongos` (page 1061) will release a connection into the thread pool *after* each read operation or command.

> **Warning:** For applications that do not use the *default* (page 396), *journaled* (page 396), or *replica acknowledged* (page 396) write concern modes of the driver, `releaseConnectionsAfterResponse` will affect the meaning of `getLastError` (page 861).
> If an application allows read operations in between write operations and `getLastError` (page 861) calls, the resulting `getLastError` (page 861) will **not** report on the success of the proceeding write operation. Use with caution.

To enable, use the following command while connected to a `mongos` (page 1061):

```
use admin
db.runCommand( { setParameter: 1, releaseConnectionsAfterResponse: true } )
```

Alternately, you may start the `mongos` (page 1061) instance with the following run-time option:

```
mongos --setParameter releaseConnectionsAfterResponse=true
```

To change this policy for the entire cluster, you must set `releaseConnectionsAfterResponse` on each `mongos` (page 1061) instance in the cluster.

CHAPTER 57

Internal Metadata and Reporting

57.1 System Collections

57.1.1 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 478), specifically for replication purposes.

57.1.2 Collections

System collections include these collections stored directly in the database:

`<database>.system.`**`namespaces`**
The `<database>.system.namespaces` (page 1133) 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 1133) collection lists all the indexes in the database. Add and remove data from this collection via the `ensureIndex()` (page 949) and `dropIndex()`

`<database>.system.`**`profile`**
The `<database>.system.profile` (page 1133) collection stores database profiling information. For information on profiling, see *Database Profiling* (page 163).

`<database>.system.users`
The `<database>.system.users` (page 238) collection stores credentials for users who have access to the database. For more information on this collection, see *Add a User to a Database* (page 226) and `<database>.system.users` (page 238).

`<database>.system.`**`js`**
The `<database>.system.js` (page 1133) collection holds special JavaScript code for use in *server side JavaScript* (page 581). See *Store a JavaScript Function on the Server* (page 582) for more information.

57.2 Database Profiler Output

The database profiler captures data information about read and write operations, cursor operations, and database commands. To configure the database profile and set the thresholds for capturing profile data, see the *Analyze Performance of Database Operations* (page 175) section.

The database profiler writes data in the `system.profile` (page 1133) collection, which is a *capped collection*. To view the profiler's output, use normal MongoDB queries on the `system.profile` (page 1133) collection.

Note: Because the database profiler writes data to the `system.profile` (page 1133) collection in a database, the profiler will profile some write activity, even for databases that are otherwise read-only.

57.2.1 Example `system.profile` Document

The documents in the `system.profile` (page 1133) collection have the following form. This example document reflects an update operation:

```
{
    "ts" : ISODate("2012-12-10T19:31:28.977Z"),
    "op" : "update",
    "ns" : "social.users",
    "query" : {
        "name" : "jane"
    },
    "updateobj" : {
        "$set" : {
            "likes" : [
                "basketball",
                "trekking"
            ]
        }
    },
    "nscanned" : 8,
    "moved" : true,
    "nmoved" : 1,
    "nupdated" : 1,
    "keyUpdates" : 0,
    "numYield" : 0,
    "lockStats" : {
        "timeLockedMicros" : {
            "r" : NumberLong(0),
            "w" : NumberLong(258)
        },
        "timeAcquiringMicros" : {
            "r" : NumberLong(0),
            "w" : NumberLong(7)
        }
    },
    "millis" : 0,
    "client" : "127.0.0.1",
    "user" : ""
}
```

57.2.2 Output Reference

For any single operation, the documents created by the database profiler will include a subset of the following fields. The precise selection of fields in these documents depends on the type of operation.

`system.profile.ts`
The timestamp of the operation.

`system.profile.op`
The type of operation. The possible values are:

- `insert`
- `query`
- `update`
- `remove`
- `getmore`
- `command`

`system.profile.ns`
The *namespace* the operation targets. Namespaces in MongoDB take the form of the *database*, followed by a dot (`.`), followed by the name of the *collection*.

`system.profile.query`
The query document used. See *Query Specification Documents* (page 64) for more information on these documents, and *Query Modification Operators* (page 827) for more information.

`system.profile.command`
The command operation.

`system.profile.updateobj`
The *update document* (page 65) passed in during an *update* (page 93) operation.

`system.profile.cursorid`
The ID of the cursor accessed by a `getmore` operation.

`system.profile.ntoreturn`
Changed in version 2.2: In 2.0, MongoDB includes this field for `query` and `command` operations. In 2.2, this information MongoDB also includes this field for `getmore` operations.

The number of documents the operation specified to return. For example, the `profile` (page 907) command would return one document (a results document) so the `ntoreturn` (page 1135) value would be `1`. The `limit(5)` (page 985) command would return five documents so the `ntoreturn` (page 1135) value would be `5`.

If the `ntoreturn` (page 1135) value is `0`, the command did not specify a number of documents to return, as would be the case with a simple `find()` (page 951) command with no limit specified.

`system.profile.ntoskip`
New in version 2.2.

The number of documents the `skip()` (page 990) method specified to skip.

`system.profile.nscanned`
The number of documents that MongoDB scans in the *index* (page 307) in order to carry out the operation.

In general, if `nscanned` (page 1135) is much higher than `nreturned` (page 1136), the database is scanning many objects to find the target objects. Consider creating an index to improve this.

system.profile.**moved**
If moved (page 1135) has a value of true indicates that the update operation moved one or more documents to a new location on disk. These operations take more time than in-place updates, and typically occur when documents grow as a result of document growth.

system.profile.**nmoved**
New in version 2.2.

The number of documents moved on disk by the operation.

system.profile.**nupdated**
New in version 2.2.

The number of documents updated by the operation.

system.profile.**keyUpdates**
New in version 2.2.

The number of *index* (page 307) keys the update changed in the operation. Changing an index key carries a small performance cost because the database must remove the old key and inserts a new key into the B-tree index.

system.profile.**numYield**
New in version 2.2.

The number of times the operation yielded to allow other operations to complete. Typically, operations yield when they need access to data that MongoDB has not yet fully read into memory. This allows other operations that have data in memory to complete while MongoDB reads in data for the yielding operation. For more information, see *the FAQ on when operations yield* (page 750).

system.profile.**lockStats**
New in version 2.2.

The time in microseconds the operation spent acquiring and holding locks. This field reports data for the following lock types:

- R - global read lock
- W - global write lock
- r - database-specific read lock
- w - database-specific write lock

system.profile.lockStats.**timeLockedMicros**
The time in microseconds the operation held a specific lock. For operations that require more than one lock, like those that lock the local database to update the *oplog*, then this value may be longer than the total length of the operation (i.e. millis (page 1136).)

system.profile.lockStats.**timeAcquiringMicros**
The time in microseconds the operation spent waiting to acquire a specific lock.

system.profile.**nreturned**
The number of documents returned by the operation.

system.profile.**responseLength**
The length in bytes of the operation's result document. A large responseLength (page 1136) can affect performance. To limit the size of the result document for a query operation, you can use any of the following:

- *Projections* (page 45)
- The limit() method (page 985)
- The batchSize() method (page 978)

system.profile.**millis**
The time in milliseconds for the server to perform the operation. This time does not include network time nor time to acquire the lock.

system.profile.**client**
The IP address or hostname of the client connection where the operation originates.

For some operations, such as `db.eval()` (page 1002), the client is `0.0.0.0:0` instead of an actual client.

system.profile.**user**
The authenticated user who ran the operation.

57.3 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` (page 1049) and `mongos` (page 1061) 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` (page 1049) if there is a mismatch between hostnames specified on the command line and in the `local.sources` (page 480) collection. `mongod` (page 1049) 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` (page 1049) (or `mongod.exe` (page 1072)) instance. The instance exits cleanly. Restart `mongod` (page 1049) with the *--upgrade* option to upgrade the database to the version supported by this `mongod` (page 1049) instance.

5

Returned by `mongod` (page 1049) if a `moveChunk` (page 880) operation fails to confirm a commit.

12

Returned by the `mongod.exe` (page 1072) 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` (page 1049) exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the `--port` run-time option.

49

Returned by `mongod.exe` (page 1072) or `mongos.exe` (page 1073) on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100

Returned by `mongod` (page 1049) when the process throws an uncaught exception.

CHAPTER 58

General Reference

58.1 MongoDB Limits and Thresholds

58.1.1 Synopsis

This document provides a collection of hard and soft limitations of the MongoDB system.

58.1.2 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` (page 1110) and the documentation for your *driver* (page 575) 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. Each index also counts as a namespace.

Size of Namespace File

Namespace files can be no larger than 2047 megabytes.

By default namespace files are 16 megabytes. You can configure the size using the `nssize` (page 1120) option.

Indexes

Index Size
The total size of an indexed value must be *less than* 1024 bytes. MongoDB will not add that value to an index if it is longer than 1024 bytes.

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 an index name to the `ensureIndex()` (page 949) 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

Enforce Unique Keys for Sharded Collections (page 548) for an alternate approach.

Number of Indexed Fields in a Compound Index
There can be no more than 31 fields in a compound index.

Capped Collections

Maximum Number of Documents in a Capped Collection
Changed in version 2.4.

If you specify a maximum number of documents for a capped collection using the `max` parameter to `create` (page 885), the limit must be less than 2^{32} documents. If you do not specify a maximum number of documents when creating a capped collection, there is no limit on the number of documents.

Replica Sets

Number of Members of a Replica Set
Replica sets can have no 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 392) for more information

Sharded Clusters

Operations Unavailable in Sharded Environments
The `group` (page 836) does not work with sharding. Use `mapReduce` (page 840) or `aggregate` (page 834) instead.

`db.eval()` (page 1002) is incompatible with sharded collections. You may use `db.eval()` (page 1002) with un-sharded collections in a shard cluster.

`$where` (page 797) does not permit references to the `db` object from the `$where` (page 797) function. This is uncommon in un-sharded collections.

The `$isolated` (page 822) update modifier does not work in sharded environments.

`$snapshot` (page 832) queries do not work in sharded environments.

`Sharding Existing Collection Data Size`
When enabling sharding on an existing collection, MongoDB imposes a maximum size on those collections to ensure that its possible to create chunks. The precise size of the limitation is a function of the chunk size and the data size: *MongoDB supports sharding all collections smaller than 256 gigabytes.* You may be able to enable sharding for existing collections between 256 and 400 gigabytes in size, depending on the distribution of document sizes.

Note: This limitation *only* applies to sharding collections that have existing data sets, and is *not* a limit on the size of a sharded collection.

See also:

`Unique Indexes in Sharded Collections` (page 1140)

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.

`Aggregation Sort Operation`
`$sort` (page 270) produces an error if the operation consumes 10 percent or more of RAM.

`2d Geospatial queries cannot use the $or operator`

See

`$or` (page 790) and *2d Index Internals* (page 351).

`Spherical Polygons must fit within a hemisphere.`
Any geometry specified with *GeoJSON* to `$geoIntersects` (page 801) or `$geoWithin` (page 800) queries, **must** fit within a single hemisphere. MongoDB interprets geometries larger than half of the sphere as queries for the smaller of the complementary geometries.

`Combination Limit with Multiple $in Expressions`
When using two or more `$in` (page 787) expressions, the product of the number of **distinct** elements in the `$in` (page 787) arrays must be less than 4000000. Otherwise, MongoDB will throw an exception of `"combinatorial limit of $in partitioning of result set exceeded"`.

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 1194) for more information on this change

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.)

Restrictions on Field Names

Field names cannot contain dots (i.e. `.`), dollar signs (i.e. `$`), or null characters. See *Dollar Sign Operator Escaping* (page 740) for an alternate approach.

58.2 Connection String URI Format

This document describes the URI format for defining connections between applications and MongoDB instances in the official MongoDB *drivers* (page 575).

58.2.1 Standard Connection String Format

This section describes the standard format of the MongoDB connection URI used to connect to a MongoDB database server. The format is the same for all official MongoDB drivers. For a list of drivers and links to driver documentation, see *MongoDB Drivers and Client Libraries* (page 575).

The following is the standard URI connection scheme:

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options
```

The components of this string are:

1. `mongodb://`

   A required prefix to identify that this is a string in the standard connection format.

2. `username:password@`

   Optional. If specified, the client will attempt to log in to the specific database using these credentials after connecting to the `mongod` (page 1049) instance.

3. `host1`

   This the only required part of the URI. It identifies a server address to connect to. It identifies either a hostname, IP address, or UNIX domain socket.

4. `:port1`

   Optional. The default value is `:27017` if not specified.

5. `hostX`

   Optional. You can specify as many hosts as necessary. You would specify multiple hosts, for example, for connections to replica sets.

6. `:portX`

   Optional. The default value is `:27017` if not specified.

7. `http://docs.mongodb.org/manualdatabase`

   Optional. The name of the database to authenticate if the connection string includes authentication credentials in the form of `username:password@`. If `http://docs.mongodb.org/manualdatabase` is not specified and the connection string includes credentials, the driver will authenticate to the `admin` database.

8. `?options`

   Connection specific options. See *Connection String Options* (page 1143) for a full description of these options.

   If the connection string does not specify a database/ you must specify a slash (i.e. `http://docs.mongodb.org/manual`) between the last `hostN` and the question mark that begins the string of options.

Example

To describe a connection to a replica set named `test`, with the following `mongod` (page 1049) hosts:

- `db1.example.net` on port `27017` and
- `db2.example.net` on port `2500`.

You would use a connection string that resembles the following:

```
mongodb://db1.example.net,db2.example.net:2500/?replicaSet=test
```

58.2.2 Connection String Options

This section lists all connection options used in the *Standard Connection String Format* (page 1142).The options are not case-sensitive.

Connection options are pairs in the following form: `name=value`. Separate options with the ampersand (i.e. `&`) character. In the following example, a connection uses the `replicaSet` and `connectTimeoutMS` options:

```
mongodb://db1.example.net,db2.example.net:2500/?replicaSet=test&connectTimeoutMS=300000
```

Semi-colon separator for connection string arguments

To provide backwards compatibility, drivers currently accept semi-colons (i.e. `;`) as option separators.

Replica Set Option

replicaSet

Specifies the name of the *replica set*, if the `mongod` (page 1049) is a member of a replica set.

When connecting to a replica set it is important to give a seed list of at least two `mongod` (page 1049) instances. If you only provide the connection point of a single `mongod` (page 1049) instance, and omit the `replicaSet` (page 1143), the client will create a *standalone* connection.

Connection Options

ssl

`true`: Initiate the connection with SSL.

`false`: Initiate the connection without SSL.

The default value is `false`.

Note: The `ssl` (page 1143) option is not supported by all drivers. See your *driver* (page 575) documentation and the *Connect to MongoDB with SSL* (page 179) document.

connectTimeoutMS
The time in milliseconds to attempt a connection before timing out. The default is never to timeout, though different drivers might vary. See the *driver* (page 575) documentation.

socketTimeoutMS
The time in milliseconds to attempt a send or receive on a socket before the attempt times out. The default is never to timeout, though different drivers might vary. See the *driver* (page 575) documentation.

Connection Pool Options

Most drivers implement some kind of connection pooling handle this for you behind the scenes. Some drivers do not support connection pools. See your *driver* (page 575) documentation for more information on the connection pooling implementation. These options allow applications to configure the connection pool when connecting to the MongoDB deployment.

maxPoolSize
The maximum number of connections in the connection pool. The default value is `100`.

minPoolSize
The minimum number of connections in the connection pool. The default value is `0`.

Note: The `minPoolSize` (page 1144) option is not supported by all drivers. For information on your driver, see the *drivers* (page 575) documentation.

maxIdleTimeMS
The maximum number of milliseconds that a connection can remain idle in the pool before being removed and closed.

This option is not supported by all drivers.

waitQueueMultiple
A number that the driver multiples the `maxPoolSize` (page 1144) value to, to provide the maximum number of threads allowed to wait for a connection to become available from the pool. For default values, see the *MongoDB Drivers and Client Libraries* (page 575) documentation.

waitQueueTimeoutMS
The maximum time in milliseconds that a thread can wait for a connection to become available. For default values, see the *MongoDB Drivers and Client Libraries* (page 575) documentation.

Write Concern Options

Write concern (page 395) describes the kind of assurances that the program:*mongod* and the driver provide to the application regarding the success and durability of the write operation. For a full explanation of write concern and write operations in general see the: *Write Operations* (page 53):

w
Defines the level and kind of write concern, that the driver uses when calling `getLastError` (page 861). This option can take either a number or a string as a value.

option number -1 The driver will *not* acknowledge write operations and will suppress all network or socket errors.

option number 0 The driver will *not* acknowledge write operations but will pass or handle any network and socket errors that it receives to the client. If you disable write concern but enable the `getLastError` (page 861) command's `w` option, `w` overrides the `w` option.

option number 1 Provides basic acknowledgment of write operations. By specifying `1`, you require that a standalone `mongod` (page 1049) instance, or the primary for *replica sets*, acknowledge all write operations. For drivers released after the *default write concern change* (page 1223), this is the default write concern setting.

option string majority For replica sets, if you specify the special `majority` value to `w` (page 1144) option, write operations will only return successfully after a majority of the configured replica set members have acknowledged the write operation.

option number n For replica sets, if you specify a number `n` greater than 1, operations with this write concern return only after `n` members of the set have acknowledged the write. If you set `n` to a number that is greater than the number of available set members or members that hold data, MongoDB will wait, potentially indefinitely, for these members to become available.

option string tags For replica sets, you can specify a *tag set* (page 444) to require that all members of the set that have these tags configured return confirmation of the write operation. See *Replica Set Tag Set Configuration* (page 444) for more information.

`wtimeoutMS`

The time in milliseconds to wait for replication to succeed, as specified in the `w` (page 1144) option, before timing out.

`journal`

Controls whether write operations will wait until the `mongod` (page 1049) acknowledges the write operations and commits the data to the on disk *journal*.

option Boolean true Enables journal commit acknowledgment write concern. Equivalent to specifying the `getLastError` (page 861) command with the `j` option enabled.

option Boolean false Does not require that `mongod` (page 1049) commit write operations to the journal before acknowledging the write operation. This is the default option for the `journal` (page 1145) parameter.

If you set `journal` (page 1145) to `true`, and specify a `w` (page 1144) value less than 1, `journal` (page 1145) prevails.

If you set `journal` (page 1145) to true, and the `mongod` (page 1049) does not have journaling enabled, as with `nojournal` (page 1120), then `getLastError` (page 861) will provide basic receipt acknowledgment (i.e. `w:1`), and will include a `jnote` field in its return document.

Read Preference Options

Read preferences (page 398) describe the behavior of read operations with regards to *replica sets*. These parameters allow you to specify read preferences on a per-connection basis in the connection string:

`readPreference`

Specifies the *replica set* read preference for this connection. This setting overrides any `slaveOk` value. The read preference values are the following:

- `primary` (page 400)
- `primaryPreferred` (page 400)
- `secondary` (page 400)

- secondaryPreferred (page 400)
- nearest (page 400)

For descriptions of each value, see *Read Preference Modes* (page 399).

The default value is `primary` (page 400), which sends all read operations to the replica set's *primary*.

readPreferenceTags

Specifies a tag set as a comma-separated list of colon-separated key-value pairs. For example:

```
dc:ny,rack:1
```

To specify a *list* of tag sets, use multiple `readPreferenceTags`. The following specifies two tag sets and an empty tag set:

```
readPreferenceTags=dc:ny,rack:1&readPreferenceTags=dc:ny&readPreferenceTags=
```

Order matters when using multiple `readPreferenceTags`.

Miscellaneous Configuration

uuidRepresentation

option standard The standard binary representation.

option csharpLegacy The default representation for the C# driver.

option javaLegacy The default representation for the Java driver.

option pythonLegacy The default representation for the Python driver.

For the default, see the *drivers* (page 575) documentation for your driver.

Note: Not all drivers support the `uuidRepresentation` (page 1146) option. For information on your driver, see the *drivers* (page 575) documentation.

58.2.3 Examples

The following provide example URI strings for common connection targets.

Database Server Running Locally

The following connects to a database server running locally on the default port:

```
mongodb://localhost
```

`admin` Database

The following connects and logs in to the `admin` database as user `sysop` with the password `moon`:

```
mongodb://sysop:moon@localhost
```

`records` Database

The following connects and logs in to the `records` database as user `sysop` with the password `moon`:

```
mongodb://sysop:moon@localhost/records
```

UNIX Domain Socket

The following connects to a UNIX domain socket:

```
mongodb:///tmp/mongodb-27017.sock
```

Note: Not all drivers support UNIX domain sockets. For information on your driver, see the *drivers* (page 575) documentation.

Replica Set with Members on Different Machines

The following connects to a *replica set* with two members, one on `db1.example.net` and the other on `db2.example.net`:

```
mongodb://db1.example.net,db2.example.com
```

Replica Set with Members on `localhost`

The following connects to a replica set with three members running on `localhost` on ports `27017`, `27018`, and `27019`:

```
mongodb://localhost,localhost:27018,localhost:27019
```

Replica Set with Read Distribution

The following connects to a replica set with three members and distributes reads to the *secondaries*:

```
mongodb://example1.com,example2.com,example3.com/?readPreference=secondary
```

Replica Set with a High Level of Write Concern

The following connects to a replica set with write concern configured to wait for replication to succeed on at least two members, with a two-second timeout.

```
mongodb://example1.com,example2.com,example3.com/?w=2&wtimeoutMS=2000
```

58.3 MongoDB Extended JSON

MongoDB *import and export utilities* (page 166) (i.e. `mongoimport` (page 1089) and `mongoexport` (page 1093)) and MongoDB REST Interfaces render an approximation of MongoDB *BSON* documents in JSON format.

The REST interface supports three different modes for document output:

- *Strict* mode that produces output that conforms to the JSON RFC specifications.

- *JavaScript* mode that produces output that most JavaScript interpreters can process (via the `--jsonp` option)
- `mongo` (page 1066) *Shell* mode produces output that the `mongo` (page 1066) shell can process. This is "extended" JavaScript format.

MongoDB can process of these representations in REST input.

Special representations of *BSON data* in JSON format make it possible to render information that have no obvious corresponding JSON. In some cases MongoDB supports multiple equivalent representations of the same type information. Consider the following table:

BSON Data Type	Strict Mode	JavaScript Mode (via JSONP)	mongo Shell Mode	Notes
`data_binary`	`{ "$binary": "<bindata>", "$type": "<t>" }`	`{ "$binary": "<bindata>", "$type": "<t>" }`	`BinData ( <t>, <bindata> )`	`<bindata>` is the base64 representation of a binary string. `<t>` is the hexadecimal representation of a single byte that indicates the data type.
`data_date`	`{ "$date": <date> }`	`new Date( <date> )`	`new Date ( <date> )`	`<date>` is the JSON representation of a 64-bit signed integer for milliseconds since epoch UTC (unsigned before version 1.9.1).
`data_timestamp`	`{ "$timestamp": { "t": <t>, "i": <i> } }`	`{ "$timestamp": { "t": <t>, "i": <i> } }`	`Timestamp( <t>, <i> )`	`<t>` is the JSON representation of a 32-bit unsigned integer for seconds since epoch. `<i>` is a 32-bit unsigned integer for the increment.
`data_regex`	`{ "$regex": "<sRegex>", "$options": "<sOptions>" }`	`/<jRegex>/<jOptions>`	`/<jRegex>/<jOptions>`	`<sRegex>` is a string of valid JSON characters. `<jRegex>` is a string that may contain valid JSON characters and unescaped double quote (`"`) characters, but may not contain unescaped forward slash (`http://docs.mongodb.o`) characters. `<sOptions>` is a string containing the regex options represented by the letters of the alphabet. `<jOptions>` is a string that may contain only the characters 'g', 'i', 'm' and 's' (added in v1.9). Because the `JavaScript` and `mongo Shell` representations support a limited range of options, any nonconforming options will be dropped when converting to this representation.
`data_oid`	`{ "$oid": "<id>" }`	`{ "$oid": "<id>" }`	`ObjectId( "<id>" )`	`<id>` is a 24-character hexadecimal string.

58.4 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 1150) 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 1151) 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 575) 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.

58.4.1 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.

[1] Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

Use

For nearly every case where you want to store a relationship between two documents, use *manual references* (page 1150). 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 1151).

58.4.2 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.

Example

DBRef document would resemble the following:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

Consider a document from a collection that stored a DBRef in a `creator` field:

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                  "$ref" : "creators",
                  "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                  "$db" : "users"
               }
}
```

The DBRef in this example, points to a document in the `creators` collection of the `users` database that has `ObjectId("5126bc054aed4daf9e2ab772")` in its `_id` field.

Note: The order of fields in the DBRef matters, and you must use the above sequence when using a DBRef.

Support

C++ The C++ driver contains no support for DBRefs. You can transverse references manually.

C# The C# driver provides access to DBRef objects with the MongoDBRef Class and supplies the FetchDBRef Method for accessing these objects.

Java The DBRef class provides supports for DBRefs from Java.

JavaScript The `mongo` (page 1066) shell's *JavaScript* (page 944) 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 1150) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

58.5 GridFS Reference

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 1152).
- `files` stores the file's metadata. For details, see *The files Collection* (page 1153).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

See also:

GridFS (page 70) for more information about GridFS.

58.5.1 The `chunks` Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <string>,
  "files_id" : <string>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

`chunks._id`
The unique *ObjectID* of the chunk.

`chunks.files_id`
The `_id` of the "parent" document, as specified in the `files` collection.

`chunks.n`
The sequence number of the chunk. GridFS numbers all chunks, starting with 0.

`chunks.data`
The chunk's payload as a *BSON* binary type.

The `chunks` collection uses a *compound index* on `files_id` and `n`, as described in *GridFS Index* (page 71).

58.5.2 The `files` Collection

Each document in the `files` collection represents a file in the *GridFS* store. Consider the following prototype of a document in the `files` collection:

```
{
  "_id" : <ObjectID>,
  "length" : <num>,
  "chunkSize" : <num>
  "uploadDate" : <timestamp>
  "md5" : <hash>

  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

Documents in the `files` collection contain some or all of the following fields. Applications may create additional arbitrary fields:

`files._id`
The unique ID for this document. The `_id` is of the data type you chose for the original document. The default type for MongoDB documents is *BSON ObjectID*.

`files.length`
The size of the document in bytes.

`files.chunkSize`
The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 kilobytes.

`files.uploadDate`
The date the document was first stored by GridFS. This value has the `Date` type.

`files.md5`
An MD5 hash returned from the filemd5 API. This value has the `String` type.

`files.filename`
Optional. A human-readable name for the document.

`files.contentType`
Optional. A valid MIME type for the document.

`files.aliases`
Optional. An array of alias strings.

`files.metadata`
Optional. Any additional information you want to store.

58.6 Glossary

$cmd A special virtual *collection* that exposes MongoDB's *database commands*. To use database commands, see *Issue Commands* (page 179).

_id A field required in every MongoDB *document*. The `_id` field must have a unique value. You can think of the `_id` field as the document's *primary key*. If you create a new document without an `_id` field, MongoDB automatically creates the field and assigns a unique BSON *ObjectId*.

accumulator An *expression* in the *aggregation framework* that maintains state between documents in the aggregation *pipeline*. For a list of accumulator operations, see `$group` (page 269).

admin database A privileged database. Users must have access to the `admin` database to run certain administrative commands. For a list of administrative commands, see *Instance Administration Commands* (page 882).

aggregation Any of a variety of operations that reduces and summarizes large sets of data. MongoDB's `aggregate()` (page 945) and `mapReduce()` (page 963) methods are two examples of aggregation operations. For more information, see *Aggregation Framework* (page 247).

aggregation framework The set of MongoDB operators that let you calculate aggregate values without having to use *map-reduce*. For a list of operators, see *Aggregation Framework Reference* (page 263).

arbiter A member of a *replica set* that exists solely to vote in *elections*. Arbiters do not replicate data. See *Replica Set Arbiter* (page 381).

B-tree A data structure commonly used by database management systems to store indexes. MongoDB uses B-trees for its indexes.

balancer An internal MongoDB process that runs in the context of a *sharded cluster* and manages the migration of *chunks*. Administrators must disable the balancer for all maintenance operations on a sharded cluster. See *Sharded Collection Balancing* (page 510).

BSON A serialization format used to store documents and make remote procedure calls in MongoDB. "BSON" is a portmanteau of the words "binary" and "*JSON*", and BSON is a representation of JSON documents. For a detailed spec, see http://bsonspec.org/. See also *Data Type Fidelity* (page 166).

BSON types The set of types supported by the *BSON* serialization format.

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

capped collection A fixed-sized *collection* that automatically overwrites its oldest entries when it reaches its maximum size. The MongoDB *oplog* that is used in *replication* is a capped collection. See *Capped Collections* (page 578).

checksum A calculated value used to ensure data integrity. The *md5* algorithm is sometimes used as a checksum.

chunk A contiguous range of *shard key* values within a particular *shard*. Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary. MongoDB splits chunks when they grow beyond the configured chunk size, which by default is 64 megabytes. MongoDB migrates chunks when a shard contains too many chunks of a collection relative to other shards. See *Data Partitioning* (page 490) and *Sharding Mechanics* (page 508).

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 See *sharded cluster*.

collection A grouping of MongoDB *documents*. A collection is the equivalent of an *RDBMS* table. A collection exists within a single *database*. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection have a similar or related purpose. See *What is a namespace in MongoDB?* (page 735).

compound index An *index* consisting of two or more keys. See *Compound Indexes* (page 311).

config database An internal database that holds the metadata associated with a *sharded cluster*. Applications and administrators should not modify the `config` dtabase in the course of normal operation. See *Config Database* (page 555).

config server A `mongod` (page 1049) instance that stores all the metadata associated with a *sharded cluster*. A production sharded cluster requires three config servers, each on a separate machine. See *Config Servers* (page 498).

control script A simple shell script, typically located in the `/etc/rc.d` or `/etc/init.d` directory, and used by the system's initialization process to start, restart or stop a *daemon* process.

CRUD An acronym for the fundamental operations of a database: Create, Read, Update, and Delete. See *Core MongoDB Operations (CRUD)* (page 39).

CSV A text-based data format consisting of comma-separated values. This format is commonly used to exchange data between relational databases since the format is well-suited to tabular data. You can import CSV files using `mongoimport` (page 1089).

cursor A pointer to the result set of a *query*. Clients can iterate through a cursor to retrieve results. By default, cursors timeout after 10 minutes of inactivity. See *Cursors* (page 49).

daemon The conventional name for a background, non-interactive process.

data-center awareness A property that allows clients to address members in a system based on their locations. *Replica sets* implement data-center awareness using *tagging*. See *Data Center Awareness* (page 151).

database A physical container for *collections*. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.

database command A MongoDB operation, other than an insert, update, remove, or query. For a list of database commands, see *Database Commands* (page 833). To use database commands, see *Issue Commands* (page 179).

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 *Database Profiling* (page 163).

datum A set of values used to define measurements on the earth. MongoDB uses the *WGS84* datum in certain *geospatial* calculations. See *Geospatial Indexes and Queries* (page 339).

dbpath The location of MongoDB's data file storage. See `dbpath` (page 1118).

delayed member A *replica set* member that cannot become primary and applies operations at a specified delay. The 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 *Delayed Replica Set Members* (page 379).

diagnostic log A verbose log of operations stored in the *dbpath*. See `diaglog` (page 1118).

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*. See *BSON Documents* (page 61).

dot notation MongoDB uses the dot notation to access the elements of an array and to access the fields of a subdocument. See *Dot Notation* (page 62).

draining The process of removing or "shedding" *chunks* from one *shard* to another. Administrators must drain shards before removing them from the cluster. See *Remove Shards from an Existing Sharded Cluster* (page 544).

driver A client library for interacting with MongoDB in a particular language. See *MongoDB Drivers and Client Libraries* (page 575).

election The process by which members of a *replica set* select a *primary* on startup and in the event of a failure. See *Replica Set Elections* (page 389).

eventual consistency A property of a distributed system that allows changes to the system to propagate gradually. In a database system, this means that readable members are not required to reflect the latest writes at all times. In MongoDB, reads to a primary have *strict consistency*; reads to secondaries have *eventual consistency*.

expression In the context of *aggregation framework*, expressions are the stateless transformations that operate on the data that passes through a *pipeline*. See *Aggregation Framework* (page 247).

failover The process that allows a *secondary* member of a *replica set* to become *primary* in the event of a failure. See *Replica Set High Availability* (page 388).

field A name-value pair in a *document*. A document has 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 a part of an effective network security strategy. See *Firewalls* (page 209).

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. See `fsync` (page 888).

geohash A geohash value is a binary representation of the location on a coordinate grid. See *Calculation of Geohash Values for 2d Indexes* (page 351).

GeoJSON A *geospatial* data interchange format based on JavaScript Object Notation (*JSON*). GeoJSON is used in *geospatial queries* (page 339). For supported GeoJSON objects, see *Location Data* (page 339). For the GeoJSON format specification, see http://geojson.org/geojson-spec.html.

geospatial Data that relates to geographical location. In MongoDB, you may store, index, and query data according to geographical parameters. See *Geospatial Indexes and Queries* (page 339).

GridFS A convention for storing large files in a MongoDB database. All of the official MongoDB drivers support this convention, as does the `mongofiles` (page 1110) program. See *GridFS* (page 70).

hashed shard key A special type of *shard key* that uses a hash of the value in the shard key field to distribute documents among members of the *sharded cluster*. See *Hashed Index* (page 315).

haystack index A *geospatial* index that enhances searches by creating "buckets" of objects grouped by a second criterion. See *Haystack Indexes* (page 348).

hidden member A *replica set* member that cannot become *primary* and are invisible to client applications. See *Hidden Replica Set Members* (page 379).

idempotent The quality of an operation to produce the same result given the same input, whether run once or run multiple times.

index A data structure that optimizes queries. See *Indexing Overview* (page 309).

initial sync The *replica set* operation that replicates data from an existing replica set member to a new or restored replica set member. See *Initial Sync* (page 407).

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` (page 1066) to display dates. The format is: `YYYY-MM-DD HH:MM.SS.millis`.

JavaScript A popular scripting language originally designed for web browsers. The MongoDB shell and certain server-side functions use a JavaScript interpreter. See *The mongo Shell* (page 603) and *Server-side JavaScript* (page 581).

journal A sequential, binary transaction log used to bring the database into a consistent state in the event of a hard shutdown. Journaling writes data first to the journal and then to the core data files. MongoDB enables journaling by default for 64-bit builds of MongoDB version 2.0 and newer. Journal files are pre-allocated and exist as files in the data directory. See *Journaling* (page 155).

JSON JavaScript Object Notation. A human-readable, plain text format for expressing structured data with support in many programming languages. For more information, see http://www.json.org. Certain MongoDB tools render an approximation of MongoDB *BSON* documents in JSON format. See *MongoDB Extended JSON* (page 1147).

JSON document A *JSON* document is a collection of fields and values in a structured format. For sample JSON documents, see http://json.org/example.html.

JSONP *JSON* with Padding. Refers to a method of injecting JSON into applications. **Presents potential security concerns**.

legacy coordinate pairs The format used for *geospatial* data prior to MongoDB version 2.4. This format stores geospatial data as points on a planar coordinate system. See *Geospatial Indexes and Queries* (page 339).

LineString A LineString is defined by an array of two or more positions. A closed LineString with four or more positions is called a LinearRing, as described in the GeoJSON LineString specification: http://geojson.org/geojson-spec.html#linestring. To use a LineString in MongoDB, see *Store GeoJSON Objects* (page 344).

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. For information on LVM and MongoDB, see *Backup and Restore Using LVM on a Linux System* (page 139).

map-reduce A data 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. For map-reduce implementation, see *Map-Reduce* (page 291). For all approaches to aggregation, see *Aggregation Framework* (page 247).

master The database that receives all writes in a conventional master-*slave* replication. In MongoDB, *replica sets* replace master-slave replication for most use cases. For more information on master-slave replication, see *Master Slave Replication* (page 408).

md5 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*. See *filemd5* (page 888).

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. The `mongofiles` (page 1110) tool provides an option to specify a MIME type to describe a file inserted into *GridFS* storage.

mongo The MongoDB shell. The `mongo` (page 1066) process starts the MongoDB shell as a daemon connected to either a `mongod` (page 1049) or `mongos` (page 1061) instance. The shell has a JavaScript interface. See *mongo* (page 1066) and *mongo Shell Methods* (page 944).

mongod The MongoDB database server. The `mongod` (page 1049) process starts the MongoDB server as a daemon. The MongoDB server manages data requests and formats and manages background operations. See *mongod* (page 1049).

MongoDB An open-source document-based database system. "MongoDB" derives from the word "humongous" because of the database's ability to scale up with ease and hold very large amounts of data. MongoDB stores *documents* in *collections* within databases.

mongos The routing and load balancing process that acts an interface between an application and a MongoDB *sharded cluster*. See *mongos* (page 1061).

namespace The canonical name for a collection or index in MongoDB. The namespace is a combination of the database name and the name of the collection or index, like so: `[database-name].[collection-or-index-name]`. All documents belong to a namespace. See *What is a namespace in MongoDB?* (page 735).

natural order The order that a database stores documents on disk. Typically, the order of documents on disks reflects insertion order, except when a document moves internally because an update operation increases its size. In *capped collections*, documents do not move internally, and therefore insertion order and natural order are identical in capped collections. MongoDB returns documents in forward natural order for a `find()` (page 951) query with no parameters. MongoDB returns documents in reverse natural order for a `find()` (page 951) query `sorted` (page 991) with a parameter of `$natural:-1`. See `$natural` (page 833).

ObjectId A special 12-byte *BSON* type that guarantees uniqueness within the *collection*. The ObjectID is generated based on timestamp, machine ID, process ID, and a process-local incremental counter. MongoDB uses ObjectId values as the default values for *_id* fields.

operator A keyword beginning with a `$` used to express an update, complex query, or data transformation. For example, `$gt` is the query language's "greater than" operator. For available operators, see *Query, Update and Projection Operators* (page 785).

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 *Replica Set Oplog* (page 405).

ordered query plan A query plan that returns results in the order consistent with the `sort()` (page 991) order. See *Query Optimization* (page 48).

padding The extra space allocated to document on the disk to prevent moving a document when it grows as the result of `update()` (page 974) operations. See *Padding Factor* (page 55).

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. See *Padding Factor* (page 55).

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 *What are page faults?* (page 768).

partition A distributed system architecture that splits data into ranges. *Sharding* uses partitioning. See *Data Partitioning* (page 490).

passive member A member of a *replica set* that cannot become primary because its `priority` (page 475) is `0`. See *Priority 0 Replica Set Members* (page 378).

pcap A packet-capture format used by `mongosniff` (page 1106) to record packets captured from network interfaces and display them as human-readable MongoDB operations. See *Options* (page 1106).

PID A process identifier. UNIX-like systems assign a unique-integer PID to each running process. You can use a PID to inspect a running process and send signals to it. See */proc File System* (page 194).

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 A series of operations in an *aggregation* process. See *Aggregation Framework* (page 247).

Point A single coordinate pair as described in the GeoJSON Point specification: http://geojson.org/geojson-spec.html#point. To use a Point in MongoDB, see *Store GeoJSON Objects* (page 344).

Polygon An array of *LinearRing* coordinate arrays, as described in the GeoJSON Polygon specification: http://geojson.org/geojson-spec.html#polygon. For Polygons with multiple rings, the first must be the exterior ring and any others must be interior rings or holes.

MongoDB does not permit the exterior ring to self-intersect. Interior rings must be fully contained within the outer loop and cannot intersect or overlap with each other. See *Store GeoJSON Objects* (page 344).

powerOf2Sizes A per-collection setting that changes and normalizes the way MongoDB allocates space for each *document*, in an effort to maximize storage reuse and to reduce fragmentation. This is the default for *TTL Collections* (page 599). See *collMod* (page 892) and `usePowerOf2Sizes` (page 892).

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. In some cases pre-splitting expedites the initial distribution of documents in *sharded cluster* by manually dividing the collection rather than waiting for the MongoDB *balancer* to do so. See *Create Chunks (Pre-Splitting)* (page 537).

primary In a *replica set*, the primary member is the current *master* instance, which receives all write operations. See *Primary* (page **??**).

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 The *shard* that holds all the un-sharded collections. See *Primary Shard* (page 497).

priority A configurable value that helps determine which members in a *replica set* are most likely to become *primary*. See `priority` (page 475).

projection A document given to a *query* that specifies which fields MongoDB returns in the result set. See *Result Projections* (page 45). For a list of projection operators, see *Projection Operators* (page 822).

query A read request. MongoDB uses a *JSON*-like query language that includes a variety of *query operators* with names that begin with a `$` character. In the `mongo` (page 1066) shell, you can issue queries using the `find()` (page 951) and `findOne()` (page 955) methods. See *Queries in MongoDB* (page 41).

query optimizer A process that generates query plans. For each query, the optimizer generates a plan that matches the query to the index that will return results as efficiently as possible. The optimizer reuses the query plan each time the query runs. If a collection changes significantly, the optimizer creates a new query plan. See *Query Optimization* (page 48).

RDBMS Relational Database Management System. A database management system based on the relational model, typically using *SQL* as the query language.

read lock In the context of a reader-writer lock, a lock that while held allows concurrent readers but no writers. See *What type of locking does MongoDB use?* (page 749).

read preference A setting that determines how clients direct read operations. Read preference affects all replica sets, including shards. By default, MongoDB directs reads to *primaries* for *strict consistency*. However, you may also direct reads to secondaries for *eventually consistent* reads. See *Read Preference* (page 398).

record size The space allocated for a document including the padding. For more information on padding, see *Padding Factor* (page 55) and *compact* (page 890).

recovering A *replica set* member status indicating that a member is not ready to begin normal activities of a secondary or primary. Recovering members 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 *Replication* (page 367).

replication A feature allowing multiple database servers to share the same data, thereby ensuring redundancy and facilitating load balancing. See *Replication* (page 367).

replication lag The length of time between the last operation in the *primary's oplog* and the last operation applied to a particular *secondary*. In general, you want to keep replication lag as small as possible. See *Replication Lag* (page 455).

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 REST is implemented over HTTP. MongoDB provides a simple HTTP REST interface that allows HTTP clients to run commands against the server. See *REST Interface* (page 159) and *REST API* (page 210).

rollback A process that reverts writes operations to ensure the consistency of all replica set members. See *Rollbacks During Replica Set Failover* (page 393).

secondary A *replica set* member that replicates the contents of the master database. Secondary members may handle read requests, but only the *primary* members can handle write operations. See *Secondaries* (page **??**).

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. See *Secondary Indexes* (page 310).

set name The arbitrary name given to a replica set. All members of a replica set must have the same name specified with the `replSet` (page 1124) setting or the `--replSet` option.

shard A single `mongod` (page 1049) instance or *replica set* that stores some portion of a *sharded cluster's* total data set. In production, all shards should be replica sets. See *Shards* (page 496).

shard key The field MongoDB uses to distribute documents among members of a *sharded cluster*. See *Shard Keys* (page 502).

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 1061) routing processes. See *Sharded Cluster Components* (page 495).

sharding A database architecture that partitions data by key ranges and distributes the data among two or more database instances. Sharding enables horizontal scaling. See *Sharding* (page 485).

shell helper A method in the `mongo` shell that provides a more concise syntax for a *database command* (page 833). Shell helpers improve the general interactive experience. See *mongo Shell Methods* (page 944).

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. See *Replica Set Primary* (page 374).

slave A read-only database that replicates operations from a *master* database in conventional master/slave replication. In MongoDB, *replica sets* replace master/slave replication for most use cases. However, for information on master/slave replication, see *Master Slave Replication* (page 408).

split The division between *chunks* in a *sharded cluster*. See *Chunk Splits in a Sharded Cluster* (page 512).

SQL Structured Query Language (SQL) is a common special-purpose programming language used for interaction with a relational database, including access control, insertions, updates, queries, and deletions. 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. `SQL` is often used as 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 An instance of `mongod` (page 1049) that is running as a single server and not as part of a *replica set*. To convert a standalone into a replica set, see *Convert a Standalone to a Replica Set* (page 426).

strict consistency A property of a distributed system requiring that all members 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 from a primary have *strict consistency*; reads from secondary members have *eventual consistency*.

sync The *replica set* operation where members replicate data from the *primary*. Sync first occurs when MongoDB creates or restores a member, which is called *initial sync*. Sync then occurs continually to keep the member updated with changes to the replica set's data. See *Replica Set Data Synchronization* (page 406).

syslog On UNIX-like systems, a logging process that provides a uniform standard for servers and processes to submit logging information. MongoDB provides an option to send output to the host's syslog system. See `syslog` (page 1117).

tag A label applied to a replica set member or shard and used by clients to issue data-center-aware operations. For more information on using tags with replica sets and with shards, see the following sections of this manual: *Tag Sets* (page 401) and *Behavior and Operations* (page 547).

TSV A text-based data format consisting of tab-separated values. This format is commonly used to exchange data between relational databases, since the format is well-suited to tabular data. You can import TSV files using `mongoimport` (page 1089).

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 before the system deletes it or ages it out. MongoDB has a TTL collection feature. See *Expire Data from Collections by Setting TTL* (page 599).

unique index An index that enforces uniqueness for a particular field across a single collection. See *Unique Indexes* (page 314).

unordered query plan A query plan that returns results in an order inconsistent with the `sort()` (page 991) order. See *Query Optimization* (page 48).

upsert An operation that will either update the first document matched by a query or insert a new document if none matches. The new document will have the fields implied by the operation. You perform upserts with the `update` (page 974) operation. See *Upsert Parameter* (page 975).

virtual memory An application's working memory, typically residing on both disk an in physical RAM.

WGS84 The default *datum* MongoDB uses to calculate geometry over an Earth-like sphere. MongoDB uses the WGS84 datum for *geospatial* queries on *GeoJSON* objects. See the "EPSG:4326: WGS 84" specification: http://spatialreference.org/ref/epsg/4326/.

working set The collection of data that MongoDB uses regularly. This data is typically (or preferably) held in RAM. See *What is the working set?* (page 768).

write concern Specifies whether a write operation has succeeded. Write concern allows your application to detect insertion errors or unavailable `mongod` (page 1049) instances. For *replica sets*, you can configure write concern to confirm replication to a specified number of members. See *Write Concern* (page 395).

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. For more information on locks, see *FAQ: Concurrency* (page 749).

writeBacks The process within the sharding system that ensures that writes issued to a *shard* that *is not* responsible for the relevant chunk get applied to the proper shard. For related information, see *What does writebacklisten in the log mean?* (page 757) and *writeBacksQueued* (page 929).

See also:

The *genindex* may provide useful insight into the reference material in this manual.

Part XV

Release Notes

Always install the latest, stable version of MongoDB. See *MongoDB Version Numbers* (page 1225) for more information.

See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

Current Stable Release

(*2.4-series*)

59.1 Release Notes for MongoDB 2.4

See the `full index of this page` for a complete list of changes included in 2.4.

MongoDB 2.4 was released on March 19, 2013.

59.1.1 What's New in MongoDB 2.4

MongoDB 2.4 represents hundreds of improvements and features driven by user requests. MongoDB 2.4 builds on the momentum of 2.2 by introducing new features that enable greater developer productivity, easier operations, improved performance and enhanced security. MongoDB 2.4 is available for download on MongoDB.org.

Developer Productivity

- Aggregation Framework refinements include an overhaul of the underlying engine introduced in MongoDB 2.2 making it easier to leverage real-time, in-place analytics. MongoDB 2.4 includes significant performance improvements, additional support for binary data, support for `$geoWithin` (page 800) and `$near` (page 801) geospatial queries, improved string concatenation with the new `$concat` (page 280) operator, and improved date calculation semantics.
- Geospatial enhancements support new use cases with support for polygon intersection queries (with `$geoIntersects` (page 801)), support for *GeoJSON*, and an improved spherical model. Learn more about *geospatial improvements in 2* (page 1177).

- Text Search provides a simplified, integrated approach to incorporating search functionality into apps with support for language specific stemming and stop words in 15 languages and real time indexes. Text search is beta for 2.4 and is not recommended for production use. Learn more about *text search* (page 1177).
- New update semantics for arrays with the `$push` (page 818) update operator. Applications can now use `$slice` (page 819) to maintain fixed size arrays, and use `$sort` (page 820) to maintain sorted arrays. Learn more about *capped arrays* (page 1181).
- New `$setOnInsert` (page 813) update operator supports specifying fields to add only on insert and *upsert* operations.

Ease of Operations

- Hashed indexes and shard keys provide simple, even distribution for reads and writes. Learn more about *hashed indexes and shard keys* (page 1178).
- New `serverStatus` (page 919) metrics including a working set analysis tool makes capacity planning easier for operations teams. Learn more about the new *serverStatus metrics* (page 1180) including the *working set analyzer* (page 931).
- More control for operators with the ability to terminate indexing operations with automatic resource cleanup.

Improved Performance

- V8 JavaScript engine offers better performance and concurrency with JavaScript based actions including those using the `$where` (page 797) query operator as well as `mapReduce` (page 840) and `eval` (page 862). Learn more about *MongoDB on V8* (page 1181), and *JavaScript Changes in MongoDB 2.4* (page 1181).
- Improvements to `count` (page 834) provide dramatically faster count operations. Counting is now up to 20 times faster for low cardinality index based counts.
- Significant optimizations to `$elemMatch` (page 809) when using a multi-key index.

More Robust Security

- Role-Based privileges allow organizations to assign more granular security policies for server, database and cluster administration. Learn more about *role based access control in MongoDB* (page 233).
- Kerberos authentication mechanism in MongoDB Enterprise.

MongoDB Enterprise

MongoDB Enterprise is a commercial edition of MongoDB that includes enterprise-grade capabilities, such as advanced security features, management tools, software integrations and certifications. Available as part of the MongoDB Enterprise Subscription, this edition includes 10gen's most comprehensive SLA and a commercial license. Continue reading for more information on MongoDB Enterprise

Learning More

These features represent only a small portion of the improvements made in MongoDB 2.4. For more details see the *MongoDB 2.4 release notes* (page 1167) and Jira for a complete list of all cases closed for MongoDB 2.4 sorted by user votes

59.1.2 Platform Support

For OS X, MongoDB 2.4 only supports OS X versions 10.6 (Snow Leopard) and later. There are no other platform support changes in MongoDB 2.4. See the downloads page for more information on platform support.

59.1.3 Upgrade Process

Upgrade MongoDB to 2.4

In the general case, the upgrade from MongoDB 2.2 to 2.4 is a binary-compatible "drop-in" upgrade: shut down the `mongod` (page 1049) instances and replace them with `mongod` (page 1049) instances running 2.4. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for *upgrading sharded clusters* (page 1170) and the considerations for *reverting to 2.2 after running 2.4* (page 1174).

Content

Upgrade Recommendations and Checklist

When upgrading, consider the following:

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` (page 1049) instance or instances.
- To upgrade to 2.4 sharded clusters *must* upgrade following the *meta-data upgrade procedure* (page 1170).
- If you're using 2.2.0 and running with `auth` (page 1118) enabled, you will need to upgrade first to 2.2.1 and then upgrade to 2.4. See *Rolling Upgrade Limitation for 2.2.0 Deployments Running with auth Enabled* (page 1174).
- If you have `system.users` (page 238) documents (i.e. for `auth` (page 1118)) that you created before 2.4 you *must* ensure that there are no duplicate values for the `user` field in the `system.users` (page 238) collection in *any* database. If you *do* have documents with duplicate user fields, you must remove them before upgrading.

  See *Compatibility Change: User Uniqueness Enforced* (page 1179) for more information.

Upgrade Standalone `mongod` Instance to MongoDB 2.4

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page. See *Install MongoDB* (page 3) for more information.
2. Shutdown your `mongod` (page 1049) instance. Replace the existing binary with the 2.4 `mongod` (page 1049) binary and restart `mongod` (page 1049).

Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4

You can upgrade to 2.4 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` (page 1049) and replacing the 2.2 binary with the 2.4 binary. After upgrading a `mongod` (page 1049) instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` (page 1017) in the `mongo` (page 1066) shell.

2. Use the `mongo` (page 1066) shell method `rs.stepDown()` (page 1018) to step down the *primary* to allow the normal *failover* (page 388) procedure. `rs.stepDown()` (page 1018) 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()` (page 1017), shut down the previous primary and replace `mongod` (page 1049) binary with the 2.4 binary and start the new process.

   Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4

Important: Only upgrade sharded clusters to 2.4 if **all** members of the cluster are currently running instances of 2.2. The only supported upgrade path for sharded clusters running 2.0 is via 2.2.

Upgrading a *sharded cluster* from MongoDB version 2.2 to 2.4 (or 2.3) requires that you run a 2.4 `mongos` (page 1061) with the *--upgrade* option, described in this procedure. The upgrade process does not require downtime.

The upgrade to MongoDB 2.4 adds epochs to the meta-data for all collections and chunks in the existing cluster. MongoDB 2.2 processes are capable of handling epochs, even though 2.2 did not require them.

This procedure applies only to upgrades from version 2.2. Earlier versions of MongoDB do not correctly handle epochs.

Warning:

- Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 555) is 4 to 5 times the amount of space currently used by the *config database* (page 555) data files. Additionally, ensure that all indexes in the *config database* (page 555) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.
- While the upgrade is in progress, you cannot make changes to the collection meta-data. For example, during the upgrade, do **not** perform:
  - `sh.enableSharding()` (page 1024),
  - `sh.shardCollection()` (page 1027),
  - `sh.addShard()` (page 1022),
  - `db.createCollection()` (page 997),
  - `db.collection.drop()` (page 948),
  - `db.dropDatabase()` (page 1002),
  - any operation that creates a database, or
  - any other operation that modifies the cluster meta-data in any way. See *Sharding Reference* (page 554) for a complete list of sharding commands. Note, however, that not all commands on the *Sharding Reference* (page 554) page modifies the cluster meta-data.
- Once you upgrade to 2.4 and complete the upgrade procedure **do not** use 2.0 `mongod` (page 1049) and `mongos` (page 1061) processes in your cluster. 2.0 process may re-introduce old meta-data formats into cluster meta-data.

Note: The upgraded config database will require more storage space than before, to make backup and working copies of the `config.chunks` (page 557) and `config.collections` (page 558) collections. As always, if storage requirements increase, the `mongod` (page 1049) might need to pre-allocate additional data files. See *What tools can I use to investigate storage use in MongoDB?* (page 768) for more information.

Meta-Data Upgrade Procedure Changes to the meta-data format for sharded clusters, stored in the *config database* (page 555), require a special meta-data upgrade procedure when moving to 2.4.

Do not perform operations that modify meta-data while performing this procedure. See *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 1170) for examples of prohibited operations.

1. Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 555) is 4 to 5 times the amount of space currently used by the *config database* (page 555) data files.

   Additionally, ensure that all indexes in the *config database* (page 555) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

   To check the version of your indexes, use `db.collection.getIndexes()` (page 956).

   If any index **on the config database** is `{v:0}`, you should rebuild those indexes by connecting to the `mongos` (page 1061) and either: rebuild all indexes using the `db.collection.reIndex()` (page 970) method, or drop and rebuild specific indexes using `db.collection.dropIndex()` and then `db.collection.ensureIndex()` (page 949). If you need to upgrade the `_id` index to `{v:1}` use `db.collection.reIndex()` (page 970).

   You may have `{v:0}` indexes on other databases in the cluster.

2. Turn off the *balancer* (page 510) in the *sharded cluster*, as described in *Disable the Balancer* (page 543).

   Optional

   For additional security during the upgrade, you can make a backup of the config database using `mongodump` (page 1075) or other backup tools.

---

3. Ensure there are no version 2.0 mongod (page 1049) or mongos (page 1061) processes still active in the sharded cluster. The automated upgrade process checks for 2.0 processes, but network availability can prevent a definitive check. Wait 5 minutes after stopping or upgrading version 2.0 mongos (page 1061) processes to confirm that none are still active.

4. Start a single 2.4 mongos (page 1061) process with configdb (page 1126) pointing to the sharded cluster's *config servers* (page 498) and with the *--upgrade* option. The upgrade process happens before the process becomes a daemon (i.e. before *--fork*.)

   You can upgrade an existing mongos (page 1061) instance to 2.4 or you can start a new *mongos* instance that can reach all config servers if you need to avoid reconfiguring a production mongos (page 1061).

   Start the mongos (page 1061) with a command that resembles the following:

   ```
   mongos --configdb <config servers> --upgrade
   ```

   Without the *--upgrade* option 2.4 mongos (page 1061) processes will fail to start until the upgrade process is complete.

   The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If there are very many sharded collections or there are stale locks held by other failed processes, acquiring the locks for all collections can take seconds or minutes. See the log for progress updates.

5. When the mongos (page 1061) process starts successfully, the upgrade is complete. If the mongos (page 1061) process fails to start, check the log for more information.

   If the mongos (page 1061) terminates or loses its connection to the config servers during the upgrade, you may always safely retry the upgrade.

   However, if the upgrade failed during the short critical section, the mongos (page 1061) will exit and report that the upgrade will require manual intervention. To continue the upgrade process, you must follow the *Resync after an Interruption of the Critical Section* (page 1173) procedure.

   Optional

   If the mongos (page 1061) logs show the upgrade waiting for the upgrade lock, a previous upgrade process may still be active or may have ended abnormally. After 15 minutes of no remote activity mongos (page 1061) will force the upgrade lock. If you can verify that there are no running upgrade processes, you may connect to a 2.2 mongos (page 1061) process and force the lock manually:

   ```
   mongo <mongos.example.net>
   ```

   ```
   db.getMongo().getCollection("config.locks").findOne({ _id : "configUpgrade" })
   ```

   If the process specified in the `process` field of this document is *verifiably* offline, run the following operation to force the lock.

   ```
   db.getMongo().getCollection("config.locks").update({ _id : "configUpgrade" }, { $set : { state :
   ```

   It is always more safe to wait for the mongos (page 1061) to verify that the lock is inactive, if you have any doubts about the activity of another upgrade operation. In addition to the `configUpgrade`, the mongos (page 1061) may need to wait for specific collection locks. Do not force the specific collection locks.

   ---

6. Upgrade and restart other mongos (page 1061) processes in the sharded cluster, *without* the *--upgrade* option.

   See *Complete Sharded Cluster Upgrade* (page 1174) for more information.

7. *Re-enable the balancer* (page 543). You can now perform operations that modify cluster meta-data.

Once you have upgraded, *do not* introduce version 2.0 MongoDB processes into the sharded cluster. This can reintroduce old meta-data formats into the config servers. The meta-data change made by this upgrade process will help prevent errors caused by cross-version incompatibilities in future versions of MongoDB.

Resync after an Interruption of the Critical Section During the short critical section of the upgrade that applies changes to the meta-data, it is unlikely but possible that a network interruption can prevent all three config servers from verifying or modifying data. If this occurs, the *config servers* (page 498) must be re-synced, and there may be problems starting new `mongos` (page 1061) processes. The *sharded cluster* will remain accessible, but avoid all cluster meta-data changes until you resync the config servers. Operations that change meta-data include: adding shards, dropping databases, and dropping collections.

Note: Only perform the following procedure *if* something (e.g. network, power, etc.) interrupts the upgrade process during the short critical section of the upgrade. Remember, you may always safely attempt the *meta data upgrade procedure* (page 1171).

To resync the config servers:

1. Turn off the *balancer* (page 510) in the sharded cluster and stop all meta-data operations. If you are in the middle of an upgrade process (*Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 1170)), you have already disabled the balancer.
2. Shut down two of the three config servers, preferably the last two listed in the `configdb` (page 1126) string. For example, if your `configdb` (page 1126) string is `configA:27019,configB:27019,configC:27019`, shut down `configB` and `configC`. Shutting down the last two config servers ensures that most `mongos` (page 1061) instances will have uninterrupted access to cluster meta-data.
3. `mongodump` (page 1075) the data files of the active config server (`configA`).
4. Move the data files of the deactivated config servers (`configB` and `configC`) to a backup location.
5. Create new, empty *data directories*.
6. Restart the disabled config servers with `--dbpath` pointing to the now-empty data directory and `--port` pointing to an alternate port (e.g. `27020`).
7. Use `mongorestore` (page 1079) to repopulate the data files on the disabled documents from the active config server (`configA`) to the restarted config servers on the new port (`configB:27020,configC:27020`). These config servers are now re-synced.
8. Restart the restored config servers on the old port, resetting the port back to the old settings (`configB:27019` and `configC:27019`).
9. In some cases connection pooling may cause spurious failures, as the `mongos` (page 1061) disables old connections only after attempted use. 2.4 fixes this problem, but to avoid this issue in version 2.2, you can restart all `mongos` (page 1061) instances (one-by-one, to avoid downtime) and use the `rs.stepDown()` (page 1018) method before restarting each of the shard *replica set primaries*.
10. The sharded cluster is now fully resynced; however before you attempt the upgrade process again, you must manually reset the upgrade state using a version 2.2 `mongos` (page 1061). Begin by connecting to the 2.2 `mongos` (page 1061) with the `mongo` (page 1066) shell:

    ```
    mongo <mongos.example.net>
    ```

    Then, use the following operation to reset the upgrade process:

    ```
    db.getMongo().getCollection("config.version").update({ _id : 1 }, { $unset : { upgradeState : 1
    ```

11. Finally retry the upgrade process, as in *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 1170).

Complete Sharded Cluster Upgrade After you have successfully completed the meta-data upgrade process described in *Meta-Data Upgrade Procedure* (page 1171), and the 2.4 `mongos` (page 1061) instance starts, you can upgrade the other processes in your MongoDB deployment.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all `mongos` (page 1061) instances in the cluster, in any order.
- Upgrade all 3 `mongod` (page 1049) config server instances, upgrading the *first* system in the *`mongos --configdb`* argument *last*.
- Upgrade each shard, one at a time, upgrading the `mongod` (page 1049) secondaries before running `replSetStepDown` (page 868) and upgrading the primary of each shard.

When this process is complete, you can now *re-enable the balancer* (page 543).

Rolling Upgrade Limitation for 2.2.0 Deployments Running with `auth` Enabled

MongoDB *cannot* support deployments that mix 2.2.0 and 2.4.0, or greater, components. MongoDB version 2.2.1 and later processes *can* exist in mixed deployments with 2.4-series processes. Therefore you cannot perform a rolling upgrade from MongoDB 2.2.0 to MongoDB 2.4.0. To upgrade a cluster with 2.2.0 components, use one of the following procedures.

1. Perform a rolling upgrade of all 2.2.0 processes to the latest 2.2-series release (e.g. 2.2.3) so that there are no processes in the deployment that predate 2.2.1. When there are no 2.2.0 processes in the deployment, perform a rolling upgrade to 2.4.0.
2. Stop all processes in the cluster. Upgrade all processes to a 2.4-series release of MongoDB, and start all processes at the same time.

Upgrade from 2.3 to 2.4

If you used a `mongod` (page 1049) from the 2.3 or 2.4-rc (release candidate) series, you can safely transition these databases to 2.4.0 or later; *however*, if you created `2dsphere` or `text` indexes using a `mongod` (page 1049) before v2.4-rc2, you will need to rebuild these indexes. For example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )

db.records.ensureIndex( { loc: "2dsphere" } )
db.records.ensureIndex( { records: "text" } )
```

Downgrade MongoDB from 2.4 to Previous Versions

For some cases the on-disk format of data files used by 2.4 and 2.2 `mongod` (page 1049) is compatible, and you can upgrade and downgrade if needed. However, several new features in 2.4 are incompatible with previous versions:

- `2dsphere` indexes are incompatible with 2.2 and earlier `mongod` (page 1049) instances.
- `text` indexes are incompatible with 2.2 and earlier `mongod` (page 1049) instances.
- using a `hashed` index as a shard key are incompatible with 2.2 and earlier `mongos` (page 1061) instances.
- `hashed` indexes are incompatible with 2.0 and earlier `mongod` (page 1049) instances.

Important: Collections sharded using hashed shard keys, should **not** use 2.2 `mongod` (page 1049) instances, which cannot correctly support cluster operations for these collections.

If you completed the *meta-data upgrade for a sharded cluster* (page 1170), you can safely downgrade to 2.2 MongoDB processes. **Do not** use 2.0 processes after completing the upgrade procedure.

Note: In sharded clusters, once you have completed the *meta-data upgrade procedure* (page 1170), you cannot use 2.0 `mongod` (page 1049) or `mongos` (page 1061) instances in the same cluster.

If you complete the meta-data upgrade, you can have a mixed cluster that has both 2.2 and 2.4 `mongod` (page 1049) and `mongos` (page 1061) instances, if needed. However, **do not** create `2dsphere` or `text` indexes in a cluster that has 2.2 components.

Considerations and Compatibility If you upgrade to MongoDB 2.4, and then need to run MongoDB 2.2 with the same data files, consider the following limitations.

- If you use a `hashed` index as the shard key index, which is only possible under 2.4 you will not be able to query data in this sharded collection. Furthermore, a 2.2 `mongos` (page 1061) cannot properly route an insert operation for a collections sharded using a `hashed` index for the shard key index: any data that you insert using a 2.2 `mongos` (page 1061), will not arrive on the correct shard and will not be reachable by future queries.
- If you *never* create an `2dsphere` or `text` index, you can move between a 2.4 and 2.2 `mongod` (page 1049) for a given data set; however, after you create the first `2dsphere` or `text` index with a 2.4 `mongod` (page 1049) you will need to run a 2.2 `mongod` (page 1049) with the *--upgrade* option and drop any `2dsphere` or `text` index.

Upgrade and Downgrade Procedures

Basic Downgrade and Upgrade **Except** as described below, moving between 2.2 and 2.4 is a drop-in replacement:

- stop the existing `mongod` (page 1049), using the *--shutdown* option as follows:

  ```
  mongod --dbpath /var/mongod/data --shutdown
  ```

  Replace `http://docs.mongodb.org/manualvar/mongod/data` with your MongoDB `dbpath` (page 1118).
- start the new `mongod` (page 1049) processes with the same `dbpath` (page 1118) setting, for example:

  ```
  mongod --dbpath /var/mongod/data
  ```

  Replace `http://docs.mongodb.org/manualvar/mongod/data` with your MongoDB `dbpath` (page 1118).

Downgrade to 2.2 After Creating a `2dsphere` or `text` Index If you have created `2dsphere` or `text` indexes while running a 2.4 `mongod` (page 1049) instance, you can downgrade at any time, by starting the `2.2` `mongod` (page 1049) with the *--upgrade* option as follows:

```
mongod --dbpath /var/mongod/data/ --upgrade
```

Then, you will need to drop any existing `2dsphere` or `text` indexes using `db.collection.dropIndex()`, for example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )
```

> **Warning:** *--upgrade* will run `repairDatabase` (page 895) on any database where you have created a `2dsphere` or `text` index, which will rebuild *all* indexes.

Troubleshooting Upgrade/Downgrade Operations If you do not use *--upgrade*, when you attempt to start a 2.2 `mongod` (page 1049) and you have created a `2dsphere` or `text` index, `mongod` (page 1049) will return the following message:

```
'need to upgrade database index_plugin_upgrade with pdfile version 4.6, new version: 4.5 Not upgradin
```

While running 2.4, to check the data file version of a MongoDB database, use the following operation in the shell:

```
db.getSiblingDB('<databasename>').stats().dataFileVersion
```

The major data file [1] version for both 2.2 and 2.4 is `4`, the minor data file version for 2.2 is `5` and the minor data file version for 2.4 is `6` **after** you create a `2dsphere` or `text` index.

Compatibility and Index Type Changes in MongoDB 2.4

In 2.4 MongoDB includes two new features related to indexes that users upgrading to version 2.4 must consider, particularly with regard to possible downgrade paths. For more information on downgrades, see *Downgrade MongoDB from 2.4 to Previous Versions* (page 1174).

New Index Types

In 2.4 MongoDB adds two new index types: `2dsphere` and `text`. These index types do not exist in 2.2, and for each database, creating a `2dsphere` or `text` index, will upgrade the data-file version and make that database incompatible with 2.2.

If you intend to downgrade, you should always drop all `2dsphere` and `text` indexes before moving to 2.2.

You can use the *downgrade procedure* (page 1174) to downgrade these databases and run 2.2 if needed, however this will run a full database repair (as with `repairDatabase` (page 895),) for all affected databases.

Index Type Validation

In MongoDB 2.2 and earlier you could specify invalid index types that did not exist. In these situations, MongoDB would create an ascending (e.g. `1`) index. Invalid indexes include index types specified by strings that do not refer to an existing index type, and all numbers other than `1` and `-1`. [2]

In 2.4, creating any invalid index will result in an error. Furthermore, you cannot create a `2dsphere` or `text` index on a collection if its containing database has any invalid index types. [1]

Example

If you attempt to add an invalid index in MongoDB 2.4, as in the following:

[1] The data file version (i.e. `pdfile version`) is independent and unrelated to the release version of MongoDB.

[2] In 2.4, indexes that specify a type of `"1"` or `"-1"` (the strings `"1"` and `"-1"`) will continue to exist, despite a warning on start-up. **However**, a *secondary* in a replica set cannot complete an initial sync from a primary that has a `"1"` or `"-1"` index. Avoid all indexes with invalid types.

```
db.coll.ensureIndex( { field: "1" } )
```

MongoDB will return the following error document:

```
{
  "err" : "Unknown index plugin '1' in index { field: \"1\" }"
  "code": 16734,
  "n": <number>,
  "connectionId": <number>,
  "ok": 1
}
```

See *Upgrade MongoDB to 2.4* (page 1169) for full upgrade instructions.

59.1.4 Changes

Major Features

Text Search

MongoDB 2.4 adds text search of content in MongoDB databases as a beta feature. With the new *text index* (page 318), and supporting, `text` (page 856) command you can search for text using boolean queries in data stored in MongoDB, using an index that updates in real-time and is always consistent with the data set. See *Text Search* (page 353) for more information about text search in MongoDB.

New Geospatial Indexes with GeoJSON and Improved Spherical Geometry

MongoDB adds the new `2dsphere` geospatial index in addition to the existing `2d` index. The `2dsphere` index supports improved spherical queries and supports the following GeoJSON objects:

- `Point`
- `LineString`
- `Polygon`

The `2dsphere` index supports all current geospatial *query operators* (page 799) and introduces the following new query operator for queries on GeoJSON data:

- `$geoWithin` (page 800) operator
- `$geoIntersects` (page 801) operator

The operators use the new `$geometry` (page 804) parameter.

The `$within` (page 801) operator no longer requires a geospatial index. Additionally, 2.4 deprecates the `$within` (page 801) operator. Use `$geoWithin` (page 800) operator instead.

For more information on geospatial indexes in 2.4, see:

- *Geospatial Indexes and Queries* (page 339)
- *2d Index Internals* (page 351)

New Hashed Index and Sharding with a Hashed Shard Key

To support an easy to configure and evenly distributed shard key, version 2.4 adds a new "`hashed`" index type that indexes documents using hashes of field values.

See *Hashed Index* (page 315) for documentation of `hashed` indexes, and *Hashed Shard Keys* (page 502) for documentation of hash-based sharding.

Security Improvements

New Modular Authentication System with Support for Kerberos

Note: Kerberos authentication is only present in MongoDB Enterprise Edition. To download and install MongoDB Enterprise, see *Install MongoDB Enterprise* (page 20).

In 2.4 the MongoDB Enterprise now supports authentication via a Kerberos mechanism. See *Deploy MongoDB with Kerberos Authentication* (page 228) for more information.

Also consider the following documents that address authenticating to MongoDB using Kerberos:

- Authenticate to MongoDB using Kerberos and the Java Driver
- Authenticate to MongoDB using Kerberos and the C# Driver

See

MongoDB Security Practices and Procedures (page 205).

SASL Library Change

In 2.4.4, MongoDB Enterprise uses Cyrus SASL. Earlier 2.4 Enterprise versions use GNU SASL (`libgsasl`). To upgrade to 2.4.4 MongoDB Enterprise or greater, you **must** install all package dependencies related to this change, including the appropriate Cyrus SASL `GSSAPI` library. See *Install MongoDB Enterprise* (page 20) for details of the dependencies.

Role Based Access Control and New Privilege Documents

MongoDB 2.4 introduces a role based access control system that provides more granular privileges to MongoDB users. See *User Privilege Roles in MongoDB* (page 233) for more information.

To support the new access control system, 2.4 also introduces a new format for documents in a database's `system.users` (page 238) collection. See *system.users Privilege Documents* (page 238) for more information.

Use `supportCompatibilityFormPrivilegeDocuments` (page 1131) to disable the legacy privilege documents, which MongoDB continues to support in 2.4.

Enhanced SSL Support

In 2.4, MongoDB instances can optionally require clients to provide SSL certificates signed by a Certificate Authority. You must use the MongoDB distribution that supports SSL, and your client driver must support SSL. See *Connect to MongoDB with SSL* (page 179) for more information.

Compatibility Change: User Uniqueness Enforced

2.4 now enforces uniqueness of the `user` field in user privilege documents (i.e. in the `system.users` (page 238) collection.) Previous versions of MongoDB did not enforce this requirement, and existing databases may have duplicates.

Administration Changes

`--setParameter` Option Available on the `mongos` and `mongod` Command Line

You can now use *`--setParameter`* on the command line and `setParameter` (page 1123) in the configuration file. For `mongod` (page 1049) the following options are available using `setParameter` (page 1123):

- `enableLocalhostAuthBypass` (page 1129)
- `enableTestCommands` (page 1129)
- `journalCommitInterval` (page 1129)
- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `replApplyBatchSize` (page 1130)
- `replIndexPrefetch` (page 1130)
- `supportCompatibilityFormPrivilegeDocuments` (page 1131)
- `syncdelay` (page 1131)
- `textSearchEnabled` (page 1132)
- `traceExceptions` (page 1131)

For `mongos` (page 1061) the following options are available using `setParameter` (page 1123):

- `enableLocalhostAuthBypass` (page 1129)
- `enableTestCommands` (page 1129)
- `logLevel` (page 1130)
- `logUserIds` (page 1130)
- `notablescan` (page 1130)
- `quiet` (page 1131)
- `supportCompatibilityFormPrivilegeDocuments` (page 1131)
- `syncdelay` (page 1131)
- `textSearchEnabled` (page 1132)

See *mongod Parameters* (page 1129) for full documentation of available parameters and their use.

Changes to `serverStatus` Output Including Additional Metrics

In 2.4 MongoDB adds a number of counters and system metrics to the output of the `serverStatus` (page 919) command, including:

- a *working set estimator* (page 931).
- operation counters, in `document` (page 932) and `operation` (page 933).
- record allocation, in `record` (page 933).
- thorough metrics of the replication process, in `repl` (page 934).
- metrics on the *ttl index* (page 599) documentation.

Additionally, in 2.4, the `serverStatus` (page 919) command can dynamically construct the *serverStatus* (page 919) document by excluding any top-level sections included by default, or including any top-level section not included by default (e.g. `workingSet` (page 932).)

See `db.serverStatus()` (page 1012) and `serverStatus` (page 919) for more information.

Increased Chunk Migration Write Concern

By default, all insert and delete operations that occur as part of a *chunk* migration in a *sharded cluster* will have an increased write concern, to ensure that at least one secondary acknowledges each insert and deletion operation. This change slows the potential speed of a chunk migration, but increases reliability and ensures that a large number of chunk migrations *cannot* affect the availability of a sharded cluster.

BSON Document Validation Enabled by Default for `mongod` and `mongorestore`

Starting in 2.4, MongoDB enables basic *BSON* object validation for `mongod` (page 1049) and `mongorestore` (page 1079) when writing to MongoDB data files. This prevents any client from inserting invalid or malformed BSON into a MongoDB database. For objects with a high degree of sub-document nesting this validation may have a small performance impact. `objcheck` (page 1116), which was previously disabled by default, provides this validation.

Indexing Changes

Support for Multiple Concurrent Index Builds

A single `mongod` (page 1049) instance can build multiple indexes in the background at the same time. See *building indexes in the background* (page 316) for more information on background index builds. Foreground index builds hold a database lock and must proceed one at a time.

`db.killOp()` Can Now Kill Foreground Index Builds

The `db.killOp()` (page 1009) method will now terminate a foreground index build, in addition to the other operations supported in previous versions.

Improved Validation of Index Types

Before 2.4, `mongod` (page 1049) would create an ascending scalar index (e.g. `{ a : 1 }`) when users attempted to create an index of a type that did not exist. Creating an index of an invalid index type will generate an error in 2.4.

See *Compatibility and Index Type Changes in MongoDB 2.4* (page 1176) for more information.

Interface Changes

`$setOnInsert` – New Update Operator

To set fields *only* when an `upsert` (page 974) performs an insert, use the `$setOnInsert` (page 813) operator with the `upsert` (page 974) .

Example

A collection named `coll` has no documents with `_id` equal to `1`.

The following `upsert` (page 974) operation inserts a document and applies the `$setOnInsert` (page 813) operator to set the fields `x` and `y`:

```
db.coll.update( { _id: 1 },
                { $setOnInsert: { x: 25, y: 30 } },
                { upsert: true } )
```

The newly-inserted document has the field `x` set to `25` and the field `y` set to 30:

```
{ "_id" : 1, "x" : 25, "y" : 30 }
```

Note: The `$setOnInsert` (page 813) operator performs no operation for `upserts` (page 974) that only perform an update and for `updates` (page 974) when the `upsert` option is `false`.

Limit Number of Elements in an Array

In 2.4, by using the `$push` (page 818) operator with the `$each` (page 819), the `$sort` (page 820), and the `$slice` (page 819) modifiers, you can add multiple elements to an array, sort and limit the number of elements in the modified array to maintain an array with a fixed number of elements.

See *Limit Number of Elements in an Array after an Update* (page 597) for an example where an update maintains the top three scores for a student.

See also:

The following pages provide additional information and examples:

- `$push` (page 818) operator
- `$each` (page 819) modifier
- `$sort` (page 820) modifier
- `$slice` (page 819) modifier

JavaScript Engine Changed to V8

JavaScript Changes in MongoDB 2.4 Consider the following impacts of *JavaScript Engine Changed to V8* (page 1181) in MongoDB 2.4:

Improved Concurrency Previously, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single mongod (page 1049) could only run a single JavaScript operation at a time. The switch to V8 improves concurrency by permitting multiple JavaScript operations to run at the same time.

Modernized JavaScript Implementation (ES5) The 5th edition of ECMAscript, abbreviated as ES5, adds many new language features, including:

- standardized JSON,
- strict mode,
- function.bind(),
- array extensions, and
- getters and setters.

With V8, MongoDB supports the ES5 implementation of Javascript with the following exceptions.

Note: The following features do not work as expected on documents **returned from MongoDB queries**:

- `Object.seal()` throws an exception on documents returned from MongoDB queries.
- `Object.freeze()` throws an exception on documents returned from MongoDB queries.
- `Object.preventExtensions()` incorrectly allows the addition of new properties on documents returned from MongoDB queries.
- `enumerable` properties, when added to documents returned from MongoDB queries, are not saved during write operations.

See SERVER-8216, SERVER-8223, SERVER-8215, and SERVER-8214 for more information.

For objects that have not been returned from MongoDB queries, the features work as expected.

Removed Non-Standard SpiderMonkey Features V8 does **not** support the following *non-standard* SpiderMonkey JavaScript extensions, previously supported by MongoDB's use of SpiderMonkey as its JavaScript engine.

E4X Extensions V8 does not support the *non-standard* E4X extensions. E4X provides a native XML object to the JavaScript language and adds the syntax for embedding literal XML documents in JavaScript code.

You need to use alternative XML processing if you used any of the following constructors/methods:

- `XML()`
- `Namespace()`
- `QName()`
- `XMLList()`
- `isXMLName()`

Destructuring Assignment V8 does not support the non-standard destructuring assignments. Destructuring assignment "extract[s] data from arrays or objects using a syntax that mirrors the construction of array and object literals." - Mozilla docs

---

Example

The following destructuring assignment is **invalid** with V8 and throws a `SyntaxError`:

```
original = [4, 8, 15];
var [b, ,c] = a;  // <== destructuring assignment
print(b) // 4
print(c) // 15
```

Iterator(), StopIteration(), and Generators V8 does not support Iterator(), StopIteration(), and generators.

InternalError() V8 does not support `InternalError()`. Use `Error()` instead.

for each...in Construct V8 does not support the use of for each...in construct. Use `for (var x in y)` construct instead.

Example

The following `for each (var x in y)` construct is **invalid** with V8:

```
var o = { name: 'MongoDB', version: 2.4 };

for each (var value in o) {
  print(value);
}
```

Instead, in version 2.4, you can use the `for (var x in y)` construct:

```
var o = { name: 'MongoDB', version: 2.4 };

for (var prop in o) {
  var value = o[prop];
  print(value);
}
```

You can also use the array *instance* method `forEach()` with the ES5 method `Object.keys()`:

```
Object.keys(o).forEach(function (key) {
  var value = o[key];
  print(value);
});
```

Array Comprehension V8 does not support Array comprehensions.

Use other methods such as the `Array` instance methods `map()`, `filter()`, or `forEach()`.

Example

With V8, the following array comprehension is **invalid**:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [i * i for each (i in a) if (i > 2)]
printjson(arr)
```

Instead, you can implement using the `Array` *instance* method `forEach()` and the ES5 method `Object.keys()`
:

```
var a = { w: 1, x: 2, y: 3, z: 4 }

var arr = [];
Object.keys(a).forEach(function (key) {
  var val = a[key];
  if (val > 2) arr.push(val * val);
})
printjson(arr)
```

Note: The new logic uses the `Array` *instance* method `forEach()` and not the *generic* method `Array.forEach()`; V8 does **not** support `Array` *generic* methods. See *Array Generic Methods* (page 1185) for more information.

Multiple Catch Blocks V8 does not support multiple `catch` blocks and will throw a `SyntaxError`.

Example

The following multiple catch blocks is **invalid** with V8 and will throw `"SyntaxError: Unexpected token if"`:

```
try {
  something()
} catch (err if err instanceof SomeError) {
  print('some error')
} catch (err) {
  print('standard error')
}
```

Conditional Function Definition V8 will produce different outcomes than SpiderMonkey with conditional function definitions.

Example

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```
function test () {
   if (false) {
      function go () {};
   }
   print(typeof go)
}
```

With SpiderMonkey, the conditional function outputs `undefined`, whereas with V8, the conditional function outputs `function`.

If your code defines functions this way, it is highly recommended that you refactor the code. The following example refactors the conditional function definition to work in both SpiderMonkey and V8.

```
function test () {
  var go;
  if (false) {
    go = function () {}
  }
  print(typeof go)
}
```

The refactored code outputs `undefined` in both SpiderMonkey and V8.

Note: ECMAscript prohibits conditional function definitions. To force V8 to throw an `Error`, enable strict mode.

```
function test () {
  'use strict';

  if (false) {
    function go () {}
  }
}
```

The JavaScript code throws the following syntax error:

```
SyntaxError: In strict mode code, functions can only be declared at top level or immediately within a
```

String Generic Methods V8 does not support String generics. String generics are a set of methods on the `String` class that mirror instance methods.

Example

The following use of the generic method `String.toLowerCase()` is **invalid** with V8:

```
var name = 'MongoDB';

var lower = String.toLowerCase(name);
```

With V8, use the `String` instance method `toLowerCase()` available through an *instance* of the `String` class instead:

```
var name = 'MongoDB';

var lower = name.toLowerCase();
print(name + ' becomes ' + lower);
```

With V8, use the `String` *instance* methods instead of following *generic* methods:

`String.charAt()`	`String.quote()`	`String.toLocaleLowerCase()`
`String.charCodeAt()`	`String.replace()`	`String.toLocaleUpperCase()`
`String.concat()`	`String.search()`	`String.toLowerCase()`
`String.endsWith()`	`String.slice()`	`String.toUpperCase()`
`String.indexOf()`	`String.split()`	`String.trim()`
`String.lastIndexOf()`	`String.startsWith()`	`String.trimLeft()`
`String.localeCompare()`	`String.substr()`	`String.trimRight()`
`String.match()`	`String.substring()`	

Array Generic Methods V8 does not support Array generic methods. Array generics are a set of methods on the `Array` class that mirror instance methods.

Example

The following use of the generic method `Array.every()` is **invalid** with V8:

```
var arr = [4, 8, 15, 16, 23, 42];
```

```
function isEven (val) {
   return 0 === val % 2;
}

var allEven = Array.every(arr, isEven);
print(allEven);
```

With V8, use the `Array` instance method `every()` available through an *instance* of the `Array` class instead:

```
var allEven = arr.every(isEven);
print(allEven);
```

With V8, use the `Array` *instance* methods instead of the following *generic* methods:

`Array.concat()`	`Array.lastIndexOf()`	`Array.slice()`
`Array.every()`	`Array.map()`	`Array.some()`
`Array.filter()`	`Array.pop()`	`Array.sort()`
`Array.forEach()`	`Array.push()`	`Array.splice()`
`Array.indexOf()`	`Array.reverse()`	`Array.unshift()`
`Array.join()`	`Array.shift()`	

Array Instance Method `toSource()` V8 does not support the `Array` instance method toSource(). Use the `Array` instance method `toString()` instead.

`uneval()` V8 does not support the non-standard method `uneval()`. Use the standardized JSON.stringify() method instead.

In 2.4 the default JavaScript engine in the `mongo` (page 1066) shell `mongod` (page 1049) is now V8. This change affects all JavaScript behavior including the `mapReduce` (page 840), `group` (page 836), and `eval` (page 862) commands, as well as the `$where` (page 797) query operator.

Use the new `interpreterVersion()` method in the `mongo` (page 1066) shell and the `javascriptEngine` (page 900) field in the output of `db.serverBuildInfo()` (page 1012) to determine which JavaScript engine a MongoDB binary uses.

The primary impacts of the change from the previous JavaScript engine, SpiderMonkey, to V8 are:

- improved concurrency for JavaScript operations,
- modernized JavaScript implementation, and
- removed non-standard SpiderMonkey features.

See *JavaScript Changes in MongoDB 2.4* (page 1181) for more information about all changes .

Additional Limitations for Map-Reduce and `$where` Operations

In MongoDB 2.4, `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions **cannot** access certain global functions or properties, such as `db`, that are available in the `mongo` (page 1066) shell.

When upgrading to MongoDB 2.4, you will need to refactor your code if your `map-reduce operations` (page 840), `group` (page 836) commands, or `$where` (page 797) operator expressions include any global shell functions or properties that are no longer available, such as `db`.

The following JavaScript functions and properties **are available** to `map-reduce operations` (page 840), the `group` (page 836) command, and `$where` (page 797) operator expressions in MongoDB 2.4:

Available Properties	Available Functions	

`args`
`MaxKey`
`MinKey`
	`assert()`
	`BinData()`
	`DBPointer()`
	`DBRef()`
	`doassert()`
	`emit()`
	`gc()`
	`HexData()`
	`hex_md5()`
	`isNumber()`
	`isObject()`
	`ISODate()`
	`isString()`
		`Map()`
		`MD5()`
		`NumberInt()`
		`NumberLong()`
		`ObjectId()`
		`print()`
		`printjson()`
		`printjsononeline()`
		`sleep()`
		`Timestamp()`
		`tojson()`
		`tojsononeline()`
		`tojsonObject()`
		`UUID()`
		`version()`

Improvements to the Aggregation Framework

MongoDB 2.4 introduces a number of additional functionality and improved performance for the *Aggregation Framework* (page 247). Consider the following additions in 2.4:

- `$match` (page 266) queries now support the `$geoWithin` (page 800) operator for bounded geospatial queries.
- The new `$geoNear` (page 271) pipeline stage to support geospatial queries.
- `$min` (page 275) operator only considers non-null and existing field values. If all the values for a field are `null` or are missing, the operator returns `null` for the minimum value.
- For sort operations where the `$sort` (page 270) stage immediately precedes a `$limit` (page 267) in the pipeline, the MongoDB can perform a more efficient sort that does not require keeping the entire result set in memory.
- The new `$millisecond` (page 285) operator returns the millisecond portion of a date.
- The new `$concat` (page 280) operator concatenates array of strings.

59.1.5 Additional Resources

- MongoDB Downloads.
- *What's New in MongoDB 2.4* (page 1167).
- All JIRA issues resolved in 2.4.
- All Backwards incompatible changes.
- All Third Party License Notices.

See `http://docs.mongodb.org/manualrelease-notes/2.4-changes` for an overview of all changes in 2.4.

CHAPTER 60

Previous Stable Releases

60.1 Release Notes for MongoDB 2.2

See the `full index of this page` for a complete list of changes included in 2.2.

60.1.1 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 2.4.5.

Synopsis

- `mongod` (page 1049), 2.2 is a drop-in replacement for 2.0 and 1.8.
- Check your *driver* (page 575) 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` (page 1049) instance or instances.
- For all upgrades of sharded clusters:
  - turn off the balancer during the upgrade process. See the *Disable the Balancer* (page 543) section for more information.
  - upgrade all `mongos` (page 1061) instances before upgrading any `mongod` (page 1049) 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` (page 1049) and `mongos` (page 1061) 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` (page 1049) instance. Replace the existing binary with the 2.2 `mongod` (page 1049) 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` (page 1049) and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` (page 1049) instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` (page 1017) in the `mongo` (page 1066) shell.
2. Use the `mongo` (page 1066) shell method `rs.stepDown()` (page 1018) to step down the *primary* to allow the normal *failover* (page 388) procedure. `rs.stepDown()` (page 1018) 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()` (page 1017), shut down the previous primary and replace `mongod` (page 1049) 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 543).
- Upgrade all `mongos` (page 1061) instances *first*, in any order.
- Upgrade all of the `mongod` (page 1049) config server instances using the *stand alone* (page 1190) procedure. To keep the cluster online, be sure that at all times at least one config server is up.
- Upgrade each shard's replica set, using the *upgrade procedure for replica sets* (page 1190) 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.

60.1.2 Changes

Major Features

Aggregation Framework

The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` (page 834) command exposes the aggregation framework, and the `db.collection.aggregate()` (page 945) helper in the `mongo` (page 1066) shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: *Aggregation Framework* (page 247)
- Reference: *Aggregation Framework Reference* (page 263)
- Examples: *Aggregation Framework Examples* (page 253)

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 599) 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 920) and *recordStats* (page 931) in *server status* (page 919) and see `db.currentOp()` (page 998), *mongotop* (page 1103), and *mongostat* (page 1097).

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 398) and *write concern* (page 395). For example, shard tagging can pin all "USA" data to one or more logical shards, while replica set tagging can control which `mongod` (page 1049) instances (e.g. "`production`" or "`reporting`") the application uses to service requests.

See the documentation for the following helpers in the `mongo` (page 1066) shell that support tagged sharding configuration:

- `sh.addShardTag()` (page 1023)

- `sh.addTagRange()` (page 1023)
- `sh.removeShardTag()` (page 1027)

Also, see *Tag Aware Sharding* (page 547) and *Administer and Manage Shard Tags* (page 530).

Fully Supported Read Preference Semantics

All MongoDB clients and drivers now support full *read preferences* (page 398), including consistent support for a full range of *read preference modes* (page 399) and *tag sets* (page 401). This support extends to the `mongos` (page 1061) 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` (page 1066) shell using the `readPref()` (page 989) cursor method.

Compatibility Changes

Authentication Changes

MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` (page 1061) 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 1190).

`findAndModify` Returns Null Value for Upserts that Perform Inserts

In version 2.2, for *upsert* that perform inserts with the `new` option set to `false`, `findAndModify` (page 851) commands will now return the following output:

```
{ 'ok': 1.0, 'value': null }
```

In the `mongo` (page 1066) shell, upsert `findAndModify` (page 851) operations that perform inserts (with `new` set to `false`.)only output a `null` value.

In version 2.0 these operations would return an empty document, e.g. `{ }`.

See: SERVER-6226 for more information.

`mongodump` 2.2 Output Incompatible with Pre-2.2 `mongorestore`

If you use the `mongodump` (page 1075) tool from the 2.2 distribution to create a dump of a database, you must use a 2.2 (or later) version of `mongorestore` (page 1079) to restore that dump.

See: SERVER-6961 for more information.

`ObjectId().toString()` Returns String Literal `ObjectId("...")`

In version 2.2, the `ObjectId.toString()` method returns the string representation of the *ObjectId()* (page 68) object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

The method now returns the *string* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

Previously, in version 2.0, the method would return the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str* (page 68), which holds the hexadecimal string value in both versions.

`ObjectId().valueOf()` Returns hexadecimal string

In version 2.2, the `ObjectId.valueOf()` method returns the value of the *ObjectId()* (page 68) object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

The method now returns the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

Previously, in version 2.0, the method would return the *object* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str* (page 68) attribute, which holds the hexadecimal string value in both versions.

Behavioral Changes

Restrictions on Collection Names

In version 2.2, collection names cannot:

- contain the `$`.
- be an empty string (i.e. `""`).

This change does not affect collections created with now illegal names in earlier versions of MongoDB.

These new restrictions are in addition to the existing restrictions on collection names which are:

- A collection name should begin with a letter or an underscore.
- A collection name cannot contain the null character.
- Begin with the `system.` prefix. MongoDB reserves `system.` for system collections, such as the `system.indexes` collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

Collections names may have any other valid UTF-8 string.

See the SERVER-4442 and the *Are there any restrictions on the names of Collections?* (page 742) 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` (page 1049) 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 `$elemMatch` (page 824) operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the *$elemMatch (projection)* (page 824) 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` (page 1073) instances as a Windows Service. See the *mongos.exe* (page 1073) reference and *MongoDB as a Windows Service* (page 18) and SERVER-1589 for more information.

Log Rotate Command Support

MongoDB for Windows now supports log rotation by way of the `logRotate` (page 897) 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* option to `mongodump` (page 1075), `mongodump` (page 1075) will now backup the definitions for all indexes that exist on the source database. When you attempt to restore this backup with `mongorestore` (page 1079), the target `mongod` (page 1049) will rebuild all indexes. See SERVER-808 for more information.

`mongorestore` (page 1079) now includes the *--noIndexRestore* option to provide the preceding behavior. Use *--noIndexRestore* to prevent `mongorestore` (page 1079) from building previous indexes.

`mongooplog` for Replaying Oplogs

The `mongooplog` (page 1086) tool makes it possible to pull *oplog* entries from `mongod` (page 1049) instance and apply them to another `mongod` (page 1049) instance. You can use `mongooplog` (page 1086) to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873 case and the *mongooplog* (page 1085) documentation.

Authentication Support for `mongotop` and `mongostat`

`mongotop` (page 1103) and `mongostat` (page 1098) 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*
- *mongotop --password*
- *mongostat --username*
- *mongostat --password*

Write Concern Support for `mongoimport` and `mongorestore`

`mongoimport` (page 1089) now provides an option to halt the import if the operation encounters an error, such as a network interruption, a duplicate key exception, or a write error. The *--stopOnError* option will produce an error rather than silently continue importing data. See SERVER-3937 for more information.

In `mongorestore` (page 1079), the *--w* option provides support for configurable write concern.

`mongodump` Support for Reading from Secondaries

You can now run `mongodump` (page 1075) when connected to a *secondary* member of a *replica set*. See SERVER-3854 for more information.

`mongoimport` Support for full 16MB Documents

Previously, `mongoimport` (page 1089) would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use `mongoimport` (page 1089) 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` (page 1086) and `mongodump` (page 1075) to query for specific timestamps. Consider the following `mongodump` (page 1075) 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` (page 1066) 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()` (page 1009) loads the contents of the current database's `system.js` collection into the current `mongo` (page 1066) shell session. See SERVER-1651 for more information.

Support for Bulk Inserts

If you pass an array of *documents* to the `insert()` (page 961) method, the `mongo` (page 1066) shell will now perform a bulk insert operation. See SERVER-3819 and SERVER-2395 for more information.

Note: For bulk inserts on sharded clusters, the `getLastError` (page 861) command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Operations

Support for Logging to Syslog

See the SERVER-2957 case and the documentation of the `syslog` (page 1117) run-time option or the *mongod --syslog* and *mongos --syslog* command line-options.

`touch` Command

Added the `touch` (page 896) command to read the data and/or indexes from a collection into memory. See: SERVER-2023 and `touch` (page 896) for more information.

indexCounters No Longer Report Sampled Data

indexCounters now report actual counters that reflect index use and state. In previous versions, these data were sampled. See SERVER-5784 and indexCounters for more information.

Padding Specifiable on compact Command

See the documentation of the compact (page 890) 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 (page 1049) 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 (page 869) command and new rs.syncFrom() (page 1018) helper in the mongo (page 1066) 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 (page 869) when overriding the default behavior.

Replica Set Members will not Sync from Members Without Indexes Unless `buildIndexes: false`

To prevent inconsistency between members of replica sets, if the member of a replica set has `buildIndexes` (page 475) set to `true`, other members of the replica set will *not* sync from this member, unless they also have `buildIndexes` (page 475) set to `true`. See SERVER-4160 for more information.

New Option To Configure Index Pre-Fetching during Replication

By default, when replicating options, *secondaries* will pre-fetch *Indexes* (page 307) associated with a query to improve replication throughput in most cases. The `replIndexPrefetch` (page 1124) setting and `--replIndexPrefetch` option allow administrators to disable this feature or allow the `mongod` (page 1049) 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 513)" documentation and SERVER-1506 for more information.

Migration Thresholds Modified

The *migration thresholds* (page 510) 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 510) documentation for more information.

60.1.3 Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice and the SERVER-4683 for more information.

60.1.4 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.

60.2 Release Notes for MongoDB 2.0

See the `full index of this page` for a complete list of changes included in 2.0.

60.2.1 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.

`mongoimport` (page 1089) and `mongoexport` (page 1093) 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` (page 1049) with the `--nojournal` run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` (page 1049) with journaling, you will see a delay as `mongod` (page 1049) creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` (page 1049) instances are interoperable with 1.8 `mongod` (page 1049) 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` (page 1049) instance. Replace the existing binary with the 2.0.x `mongod` (page 1049) 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` (page 1049) 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 395) in your application code to confirm that each update reaches multiple servers.
3. Use the `rs.stepDown()` (page 1018) to step down the primary to allow the normal *failover* (page 388) procedure.

   `rs.stepDown()` (page 1018) and `replSetStepDown` (page 868) 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` (page 1049) 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 1061) routers in any order.

60.2.2 Changes

Compact Command

A `compact` (page 890) 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* (page 338). Indexes are often 25% smaller and 25% faster (depends on the use case). When upgrading from previous versions, the benefits of the new index type are realized only if you create a new index or re-index an old one.

Dates are now signed, and the max index key size has increased slightly from 819 to 1024 bytes.

All operations that create a new index will result in a 2.0 index by default. For example:

- Reindexing results on an older-version index results in a 2.0 index. However, reindexing on a secondary does *not* work in versions prior to 2.0. Do not reindex on a secondary. For a workaround, see SERVER-3866.
- The `repairDatabase` command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 1200), invoke the `compact` (page 890) command.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0. See *Build Old Style Indexes* (page 338).

Sharding Authentication

Applications can now use authentication with *sharded clusters*.

Replica Sets

Hidden Nodes in Sharded Clusters

In 2.0, `mongos` (page 1061) instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, `mongos` (page 1061) if you reconfigured a member as hidden, you *had* to restart `mongos` (page 1061) to prevent queries from reaching the hidden member.

Priorities

Each *replica set* member can now have a priority value consisting of a floating-point from 0 to 1000, inclusive. Priorities let you control which member of the set you prefer to have as *primary* the member with the highest priority that can see a majority of the set will be elected primary.

For example, suppose you have a replica set with three members, `A`, `B`, and `C`, and suppose that their priorities are set as follows:

- `A`'s priority is `2`.
- `B`'s priority is `3`.
- `C`'s priority is `1`.

During normal operation, the set will always chose `B` as primary. If `B` becomes unavailable, the set will elect `A` as primary.

For more information, see the `priority` (page 475) 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 395) 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 *MongoDB Drivers and Client Libraries* (page 575) 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 395).

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 448).

Primary Checks for a Caught up Secondary before Stepping Down

To minimize time without a *primary*, the `rs.stepDown()` (page 1018) 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 441).

Extended Shutdown on the Primary to Minimize Interruption

When you call the `shutdown` (page 896) 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` (page 896) command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode

When `repair` or `compact` (page 890) 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 `$within` (page 801) queries are also now supported for simple polygon shapes. For details, see the `$within` (page 801) operator documentation.

Journaling Enhancements

- Journaling is now enabled by default for 64-bit platforms. Use the `--nojournal` command line option to disable it.
- The journal is now compressed for faster commits to disk.
- A new `--journalCommitInterval` run-time option exists for specifying your own group commit interval. The default settings do not change.
- A new `{ getLastError: { j: true } }` (page 861) 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 575), 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` (page 861) 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` (page 861) results.

See OP_INSERT.

Note: For bulk inserts on sharded clusters, the `getLastError` (page 861) command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Map Reduce

Output to a Sharded Collection

Using the new `sharded` flag, it is possible to send the result of a map/reduce to a sharded collection. Combined with the `reduce` or `merge` flags, it is possible to keep adding data to very large collections from map/reduce jobs.

For more information, see MapReduce Output Options and *mapReduce* (page 840).

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 840).

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 (page 798).

$and

A special boolean $and (page 791) query operator is now available.

Command Output Changes

The output of the validate (page 908) 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 (page 1066) 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 1066).

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. auth (page 1118)) *all* database commands require authentication, *except* the following commands.

- isMaster (page 871)
- authenticate
- getnonce
- buildInfo (page 899)
- ping (page 907)
- isdbgrid (page 881)

60.2.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 2.0
- All Backward Incompatible Changes

60.3 Release Notes for MongoDB 1.8

See the `full index of this page` for a complete list of changes included in 1.8.

60.3.1 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 1205).
- The `mapReduce` (page 840) command has changed in 1.8, causing incompatibility with previous releases. `mapReduce` (page 840) 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` (page 840) 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` (page 1049) instance.
3. Replace the existing binary with the 1.8.x `mongod` (page 1049) 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()` (page 1015) 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 1205).
   - If the shard is a single `mongod` (page 1049) process, shut it down and then restart it with the 1.8.x binary from the MongoDB Download Page.

3. For each `mongos` (page 1061):

   (a) Shut down the `mongos` (page 1061) 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* (page 155) 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.

60.3.2 Changes

Journaling

MongoDB now supports write-ahead *Journaling* (page 155) to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` (page 1049) 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 314) 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 322) enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

Incremental MapReduce Support

The `mapReduce` (page 840) 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` (page 840) document.

Additional Changes and Enhancements

1.8.1

- Sharding migrate fix when moving larger chunks.

- Durability fix with background indexing.
- Fixed mongos concurrency issue with many incoming connections.

1.8.0

- All changes from 1.7.x series.

1.7.6

- Bug fixes.

1.7.5

- *Journaling* (page 155).
- Extent allocation improvements.
- Improved *replica set* connectivity for `mongos` (page 1061).
- `getLastError` (page 861) improvements for *sharding*.

1.7.4

- `mongos` (page 1061) routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` (page 840) output options.
- *Sparse Indexes* (page 314).

1.7.3

- Initial *covered index* (page 322) support.
- Distinct can use data from indexes when possible.
- `mapReduce` (page 840) can merge or reduce results into an existing collection.
- `mongod` (page 1049) tracks and `mongostat` (page 1098) displays network usage. See *mongostat* (page 1097).
- Sharding stability improvements.

1.7.2

- `$rename` (page 810) operator allows renaming of fields in a document.
- `db.eval()` (page 1002) not to block.
- Geo queries with sharding.
- *`mongostat --discover`* option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for `$elemMatch` (page 824) on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` (page 817) works on primitives in arrays.

1.7.0

- Sharding performance improvements for heavy insert loads.
- Slave delay support for replica sets.
- `getLastErrorDefaults` (page 477) for replica sets.
- Auto completion in the shell.
- Spherical distance for geo search.
- All fixes from 1.6.1 and 1.6.2.

Release Announcement Forum Pages

- 1.8.1, 1.8.0
- 1.7.6, 1.7.5, 1.7.4, 1.7.3, 1.7.2, 1.7.1, 1.7.0

60.3.3 Resources

- MongoDB Downloads
- All JIRA Issues resolved in 1.8

60.4 Release Notes for MongoDB 1.6

See the `full index of this page` for a complete list of changes included in 1.6.

60.4.1 Upgrading

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown `mongod` (page 1049) then restart with the new binaries.

Please note that you should upgrade to the latest version of whichever driver you're using. Certain drivers, including the Ruby driver, will require the upgrade, and all the drivers will provide extra features for connecting to replica sets.

60.4.2 Sharding

Sharding (page 485) is now production-ready, making MongoDB horizontally scalable, with no single point of failure. A single instance of `mongod` (page 1049) can now be upgraded to a distributed cluster with zero downtime when the need arises.

- *Sharding* (page 485)
- *Deploy a Sharded Cluster* (page 516)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 524)

60.4.3 Replica Sets

Replica sets (page 367), which provide automated failover among a cluster of `n` nodes, are also now available.

Please note that replica pairs are now deprecated; we strongly recommend that replica pair users upgrade to replica sets.

- *Replication* (page 367)
- *Deploy a Replica Set* (page 416)
- *Convert a Standalone to a Replica Set* (page 426)

60.4.4 Other Improvements

- The `w` option (and `wtimeout`) forces writes to be propagated to `n` servers before returning success (this works especially well with replica sets)
- *$or queries* (page 790)
- Improved concurrency
- *$slice* (page 826) operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The `findAndModify` (page 851) command now supports upserts. It also allows you to specify fields to return
- $showDiskLoc option to see disk location of a document
- Support for IPv6 and UNIX domain sockets

60.4.5 Installation

- Windows service improvements
- The C++ client is a separate tarball from the binaries

60.4.6 1.6.x Release Notes

- 1.6.5

60.4.7 1.5.x Release Notes

- 1.5.8
- 1.5.7
- 1.5.6
- 1.5.5
- 1.5.4
- 1.5.3
- 1.5.2
- 1.5.1
- 1.5.0

You can see a full list of all changes on JIRA.

Thank you everyone for your support and suggestions!

60.5 Release Notes for MongoDB 1.4

See the `full index of this page` for a complete list of changes included in 1.4.

60.5.1 Upgrading

We're pleased to announce the 1.4 release of MongoDB. 1.4 is a drop-in replacement for 1.2. To upgrade you just need to shutdown `mongod` (page 1049), then restart with the new binaries. (Users upgrading from release 1.0 should review the *1.2 release notes* (page 1214), in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

60.5.2 Core Server Enhancements

- *concurrency* (page 749) improvements
- indexing memory improvements
- *background index creation* (page 316)

- better detection of regular expressions so the index can be used in more cases

60.5.3 Replication and Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (`--fastsync`)
- configurable slave delay (`--slavedelay`)
- replication handles clock skew on master
- *$inc* (page 810) replication fixes
- sharding alpha 3 - notably 2-phase commit on config servers

60.5.4 Deployment and Production

- *configure "slow threshold" for profiling* (page 175)
- ability to do *fsync + lock* (page 888) for backing up raw files
- option for separate directory per db (`--directoryperdb`)
- `http://localhost:28017/_status` to get serverStatus via http
- REST interface is off by default for security (`--rest` to enable)
- can rotate logs with a db command, *logRotate* (page 897)
- enhancements to *serverStatus* (page 919) command (db.serverStatus()) - counters and *replication lag* (page 455) stats
- new *mongostat* (page 1097) tool

60.5.5 Query Language Improvements

- *$all* (page 785) with regex
- *$not* (page 792)
- partial matching of array elements *$elemMatch* (page 824)
- $ operator for updating arrays
- *$addToSet* (page 815)
- *$unset* (page 814)
- *$pull* (page 817) supports object matching
- *$set* (page 814) with array indexes

60.5.6 Geo

- *2d geospatial search* (page 351)
- geo *$center* (page 804) and *$box* (page 805) searches

60.6 Release Notes for MongoDB 1.2.x

See the `full index of this page` for a complete list of changes included in 1.2.

60.6.1 New Features

- More indexes per collection
- Faster index creation
- Map/Reduce
- Stored JavaScript functions
- Configurable fsync time
- Several small features and fixes

60.6.2 DB Upgrade Required

There are some changes that will require doing an upgrade if your previous version is <= 1.0.x. If you're already using a version >= 1.1.x then these changes aren't required. There are 2 ways to do it:

- `--upgrade`
  - stop your `mongod` (page 1049) process
  - run `./mongod --upgrade`
  - start `mongod` (page 1049) again
- use a slave
  - start a slave on a different port and data directory
  - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

60.6.3 Replication Changes

- There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

60.6.4 mongoimport

- `mongoimportjson` has been removed and is replaced with *mongoimport* (page 1089) that can do json/csv/tsv

60.6.5 field filter changing

- We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter only changes the output, not which objects are returned. If you need that behavior, you can use *$exists* (page 794)

CHAPTER 61

Current Development Series

61.1 Release Notes for MongoDB 2.6 (Development Series 2.5.x)

MongoDB 2.6 is currently in development, as part of the 2.5 development release series. While 2.5-series releases are currently available, these versions of MongoDB, including the 2.6 release candidate builds, are for **testing only and not for production use**.

This document will eventually contain the full release notes for MongoDB 2.6; before its release this document covers the 2.5 development series as a work-in-progress.

See the `full index of this page` for a complete list of changes included in 2.6 (Development Series 2.5.x).

61.1.1 Downloading

You can download the 2.6 release candidate on the downloads page in the *Development Release (Unstable)* section. There are no distribution packages for development releases, but you can use the binaries provided for testing purposes. See *Install MongoDB on Linux* (page 11), *Install MongoDB on Windows* (page 16), or *Install MongoDB on OS X* (page 13) for the basic installation process.

61.1.2 Changes

SASL Library Change

MongoDB Enterprise uses Cyrus SASL instead of GNU SASL (`libgsasl`). This change has the following SASL2 and Cyrus SASL library and GSSAPI plugin dependencies:

For Debian or Ubuntu, install the following:

```
sudo apt-get install cyrus-sasl2-dbg cyrus-sasl2-mit-dbg libsasl2-2 libsasl2-dev libsasl2-modules lib
```

For CentOS, Red Hat Enterprise Linux, and Amazon AMI, install the following:

```
sudo yum install cyrus-sasl cyrus-sasl-lib cyrus-sasl-devel cyrus-sasl-gssapi
```

For SUSE, install the following:

```
sudo zypper install cyrus-sasl cyrus-sasl-devel cyrus-sasl-gssapi
```

LDAP Support for Authentication

MongoDB Enterprise provides support for proxy authentication of users. This change allows administrators to configure a MongoDB cluster to authenticate users via Linux PAM or by proxying authentication requests to a specified LDAP service.

> **Warning:** Because this change uses `SASL PLAIN` mechanism to transmit the user password to the MongoDB server, you should, in general, use only on a trusted channel (VPN, SSL, trusted wired network).

Configuration

LDAP support for user authentication requires proper configuration of the `saslauthd` daemon process as well as introduces a new server parameter, `saslauthdPath`. `saslauthdPath` is the path to the Unix Domain Socket of the `saslauthd` instance to use for proxy authentication.

`saslauthd` Configuration On systems that configure `saslauthd` with a `/etc/sysconfig/saslauthd` file, such as Red Hat Enterprise Linux, Fedora, CentOS, Amazon Linux AMI, set the mechanism `MECH` to `ldap`:

```
MECH=ldap
```

On systems that configure `saslauthd` with a `/etc/default/saslauthd` file, set the mechanisms option to `ldap`:

```
MECHANISMS="ldap"
```

To use with *ActiveDirectory*, start `saslauthd` with the following configuration options:

```
ldap_servers: <ldap uri, e.g. ldaps://ad.example.net>
ldap_use_sasl: yes
ldap_mech: DIGEST-MD5
ldap_auth_method: fastbind
```

To connect to an OpenLDAP server, use a test `saslauthd.conf` with the following content:

```
ldap_servers: <ldap uri, e.g. ldaps://ad.example.net>
ldap_search_base: ou=Users,dc=example,dc=com
ldap_filter: (uid=%u)
```

To use this sample OpenLDAP configuration, create users with a `uid` attribute (login name) and place under the `Users` organizational unit (`ou`).

To test the `saslauthd` configuration, use `testsaslauthd` utility, as in the following example:

```
testsaslauthd -u testuser -p testpassword -s mongod -f /var/run/saslauthd/mux
```

For more information on `saslauthd` configuration, see http://www.openldap.org/doc/admin24/guide.html#Configuringsaslauthd.

MongoDB Server Configuration Configure the MongoDB server with the authenticationMechanisms parameter and the saslauthdPath parameters using either the command line option *--setParameter* or the *configuration file* (page 1115):

- If saslauthd has a socket path of /<some>/<path>/saslauthd, set the saslauthdPath parameter to /<some>/<path>/saslauthd/mux and the authenticationMechanisms parameter to PLAIN, as in the following command line example:

  ```
  mongod --setParameter saslauthdPath=/<some>/<path>/saslauthd/mux --setParameter authenticationMe
  ```

  Or to set the configuration in the *configuration file* (page 1115), add the parameters:

  ```
  setParameter=saslauthdPath=/<some>/<path>/saslauthd/mux
  setParameter=authenticationMechanisms=PLAIN
  ```

- Otherwise, set the saslauthdPath to the empty string "" to use the library's default value and the authenticationMechanisms parameter to PLAIN, as in the following command line example:

  ```
  mongod --setParameter saslauthdPath="" --setParameter authenticationMechanisms=PLAIN
  ```

  Or to set the configuration in the *configuration file* (page 1115), add the parameters:

  ```
  setParameter=saslauthdPath=""
  setParameter=authenticationMechanisms=PLAIN
  ```

Authenticate in the mongo Shell

To use this authentication mechanism in the mongo (page 1066) shell, you **must** pass digestPassword: false to db.auth() (page 995) when authenticating on the $external database, since the server must receive an undigested password to forward on to saslauthd, as in the following example:

```
use $external
db.auth(
         {
           mechanism: "PLAIN",
           user: "application/reporting@EXAMPLE.NET",
           pwd: "some1nterestingPwd",
           digestPassword: false
         }
       )
```

x.509 Authentication

MongoDB introduces x.509 certificate authentication for use with a secure *SSL connection* (page 179).

Important: To use SSL, you must either use MongoDB Enterprise or build MongoDB locally using scons with the --ssl option.

The x.509 authentication allows clients to authenticate to servers with certificates instead of with username and password.

The x.509 authentication also allows sharded cluster members and replica set members to use x.509 certificates to verify their membership to the cluster or the replica set instead of using key files. The membership authentication is an internal process.

x.509 Certificate

The x.509 certificate for client authentication and the x.509 certificate for internal authentication have different properties.

The client certificate must have the following properties:

- A single Certificate Authority (CA) must issue the certificates for both the client and the server.
- Client certificates must contain the following fields:

```
keyUsage = digitalSignature
extendedKeyUsage = clientAuth
```

The member certificate, used for internal authentication to verify membership to the sharded cluster or a replica set, must have the following properties:

- A single Certificate Authority (CA) must issue all the x.509 certificates for the members of a sharded cluster or a replica set.
- The member certificate's `subject`, which contains the Distinguished Name (`DN`), must match the `subject` of the certificate on the server, *starting from and including* the Organizational Unit (`OU`) of the certificate on the server.

New Protocol and Parameters

The change for x.509 authentication introduces a new `MONGODB-X509` protocol. For internal authentication for membership, the change also introduces the `--clusterAuthMode`, `--sslClusterFile` and the `--sslClusterPassword` options.

Use the `--clusterAuthMode` option to enable internal x.509 authentication for membership. The `--clusterAuthMode` option can have one of the following values:

Value	Description
`keyfile`	Default value. Use keyfile for authentication.
`sendKeyfile`	For rolling upgrade purposes. Send the keyfile for authentication but can accept either keyfile or x.509 certificate.
`sendX509`	For rolling upgrade purposes. Send the x.509 certificate for authentication but can accept either keyfile or x.509 certificate.
`x509`	Recommended. Send the x.509 certificate for authentication and accept **only** x.509 certificate.

For the `--sslClusterFile` option, specify the full path to the x.509 certificate and key PEM file for the cluster or set member. If the key is encrypted, specify the password with the `--sslClusterPassword` option.

Configure MongoDB Server to Use x.509

Configure the MongoDB server from the command line, as in the following [1]:

```
mongod --sslOnNormalPorts --sslPEMKeyFile <path to sslCertificate and key PEM file> --sslCAFile <path
```

You may also specify these options in the *configuration file* (page 1115):

```
sslOnNormalPorts = true
sslPEMKeyFile = <path to sslCertificate and key PEM file>
sslCAFile = <path to the root CA PEM file>
```

[1] Include any additional options, SSL or otherwise, that are required for your specific configuration.

To specify the x.509 certificate for internal cluster member authentication, append the additional SSL options `--clusterAuthMode` and `--sslClusterFile`, as in the following example for a member of a replica set [1]:

```
mongod --replSet <name> --sslOnNormalPorts --sslPEMKeyFile <path to sslCertificate and key PEM file>
```

Authenticate with a x.509 Certificate

To authenticate with a client certificate, you must first add a MongoDB user that corresponds to the client certificate. See *Add x.509 Certificate subject as a User* (page 1221).

To authenticate, use the `db.auth()` (page 995) method in the `$external` database. For the `mechanism` field, specify `"MONGODB-X509"`, and for the `user` field, specify the user, or the `subject`, that corresponds to the client certificate.

For example, if using the `mongo` (page 1066) shell,

1. Connect `mongo` (page 1066) shell to the `mongod` (page 1049) set up for SSL:

   ```
   mongo --ssl --sslPEMKeyFile <path to CA signed client PEM file>
   ```

2. To perform the authentication, use the `db.auth()` (page 995) method in the `$external` database.

   ```
   db.getSiblingDB("$external").auth(
                                      {
                                        mechanism: "MONGODB-X509",
                                        user: "CN=myName,OU=myOrgUnit,O=myOrg,L=myLocality,ST=mySta
                                      }
                                    )
   ```

Add x.509 Certificate `subject` as a User

To authenticate with a client certificate, you must first add the value of the `subject` from the client certificate as a MongoDB user.

1. You can retrieve the `subject` from the client certificate with the following command:

   ```
   openssl x509 -in <pathToClient PEM> -inform PEM -subject -nameopt RFC2253
   ```

   The command returns the `subject` string as well as certificate:

   ```
   subject= CN=myName,OU=myOrgUnit,O=myOrg,L=myLocality,ST=myState,C=myCountry
   -----BEGIN CERTIFICATE-----
   # ...
   -----END CERTIFICATE-----
   ```

2. Add the value of the `subject`, omitting the spaces, from the certificate as a user. For example, in the `mongo` (page 1066) shell, to add the user to the `test` database:

   ```
   use test
   db.addUser({
               user: "CN=myName,OU=myOrgUnit,O=myOrg,L=myLocality,ST=myState,C=myCountry",
               userSource: '$external',
               roles: ['readAnyDatabase', 'readWriteAnyDatabase']
             })
   ```

See *Add a User to a Database* (page 226) for details on adding a user with roles using *privilege documents* (page 238).

Upgrade Clusters to x.509 Authentication

To upgrade clusters that are currently using keyfile authentication to x.509 authentication, use a rolling upgrade process:

1. For each node of a cluster, set `--clusterAuthMode` to `sendKeyFile`. With this setting, each node continues to use its keyfile to authenticate itself as a member. However, each node can now accept either a keyfile or the x.509 certificate from other members to authenticate those members. Upgrade all nodes of the cluster to this setting.
2. Then, for each node of a cluster, set `--clusterAuthMode` to `sendX509` and set `--sslClusterFile` to the appropriate path of the node's certificate. [2] With this setting, each node uses its x.509 certificate to authenticate itself as a member. However, each node continues to accept either a keyfile or the x.509 certificate from other members to authenticate those members. Upgrade all nodes of the cluster to this setting.
3. Optional but recommended. Finally, for each node of the cluster, set `--clusterAuthMode` to `x509` to only use the x.509 certificate for authentication.

Background Index Builds Replicate to Secondaries

Starting in MongoDB 2.5.0, if you initiate a *background index build* (page 316) on a *primary*, the secondaries will replicate the index build in the background. In previous versions of MongoDB, secondaries built all indexes in the foreground, even if the primary built an index in the background.

For all index builds, secondaries will not begin building indexes until the primary has successfully completed the index build.

`mongod` Automatically Continues in Progress Index Builds Following Restart

If your `mongod` (page 1049) instance was building an index when it shutdown or terminated, `mongod` (page 1049) will now continue building the index when the `mongod` (page 1049) restarts. Previously, the index build *had* to finish building before `mongod` (page 1049) shutdown.

To disable this behavior the 2.5 series adds a new run time option, `noIndexBuildRetry` (page 1222) (or via, `--noIndexBuildRetry` on the command line,) for `mongod` (page 1049). `noIndexBuildRetry` (page 1222) prevents `mongod` (page 1049) from continuing rebuilding indexes that did not finished building when the `mongod` (page 1049) last shut down.

`noIndexBuildRetry`

By default, `mongod` (page 1049) will attempt to rebuild indexes upon start-up *if* `mongod` (page 1049) shuts down or stops in the middle of an index build. When enabled, this option prevents this behavior.

[2] If the key is encrypted, set the `--sslClusterPassword` to the password to decrypt the key.

Other MongoDB Release Notes

62.1 Default Write Concern Change

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

62.1.1 Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.

   This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.

   The new top-level connection class is named `MongoClient`, or similar depending on how host languages handle namespacing.

2. The default write concern on the new `MongoClient` class will be to acknowledge all write operations [1]. This will allow your application to receive acknowledgment of all write operations.

   See the documentation of *Write Concern* (page 395) for more information about write concern in MongoDB.

   Please migrate to the new `MongoClient` class expeditiously.

62.1.2 Releases

The following driver releases will include the changes outlined in *Changes* (page 1223). See each driver's release notes for a full account of each release as well as other related driver-specific changes.

- C#, version 1.7
- Java, version 2.10.0
- Node.js, version 1.2
- Perl, version 0.501.1

[1] The drivers will call `getLastError` (page 861) without arguments, which is logically equivalent to the `w: 1` option; however, this operation allows *replica set* users to override the default write concern with the `getLastErrorDefaults` (page 477) setting in the *Replica Set Configuration* (page 473).

- PHP, version 1.4
- Python, version 2.4
- Ruby, version 1.8

CHAPTER 63

MongoDB Version Numbers

For MongoDB `2.4.1`, `2.4` refers to the release series and `.1` refers to the revision. The second component of the release series (e.g. `4` in `2.4.1`) describes the type of release series. Release series ending with even numbers (e.g. `4` above) are *stable* and ready for production, while odd numbers are for *development* and testing only.

Generally, changes in the release series (e.g. `2.4`) mark the introduction of new features that may break backwards compatibility. Changes to the revision number mark the release bug fixes and backwards-compatible changes.

Important: Always upgrade to the latest stable revision of your release series.

The version numbering system for MongoDB differs from the system used for the MongoDB drivers. Drivers use only the first number to indicate a major version. For details, see *Driver Version Numbers* (page 575).

Example

Version numbers

- 2.0.0 : Stable release.
- 2.0.1 : Revision.
- 2.1.0 : Development release *for testing only*. Includes new features and changes for testing. Interfaces and stability may not be compatible in development releases.
- 2.2.0 : Stable release. This is a culmination of the 2.1.x development series.

Part XVI

About MongoDB Documentation

The MongoDB Manual contains comprehensive documentation on the MongoDB *document*-oriented database management system. This page describes the manual's licensing, editions, and versions, and describes how to make a change request and how to contribute to the manual.

For more information on MongoDB, see MongoDB: A Document Oriented Database. To download MongoDB, see the downloads page.

CHAPTER 64

License

CHAPTER 65

Editions

In addition to the MongoDB Manual, you can also access this content in the following editions:

- ePub Format
- Single HTML Page
- PDF Format
- HTML tar.gz

You also can access PDF files that contain subsets of the MongoDB Manual:

- MongoDB Reference Manual
- MongoDB Use Cases
- MongoDB CRUD Operations
- Replication and MongoDB
- Sharding and MongoDB

MongoDB Reference documentation is also available as part of dash.

CHAPTER 66

Version and Revisions

This version of the manual reflects version 2.4 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 "4b5853333955861e606c3557d2d65c31ea361211" 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/".

CHAPTER 67

Report an Issue or Make a Change Request

To report an issue with this manual or to make a change request, file a ticket at the MongoDB DOCS Project on Jira.

CHAPTER 68

Contribute to the Documentation

68.1 MongoDB Manual Translation

The original authorship language for all MongoDB documentation is American English. However, ensuring that speakers of other languages can read and understand the documentation is of critical importance to the documentation project.

In this direction, the MongoDB Documentation project uses the service provided by Smartling to translate the MongoDB documentation into additional non-English languages. This translation project is largely supported by the work of volunteer translators from the MongoDB community who contribute to the translation effort.

If you would like to volunteer to help translate the MongoDB documentation, please:

- complete the 10gen/MongoDB Contributor Agreement, and
- create an account on Smartling at translate.docs.mongodb.org.

Please use the same email address you use to sign the contributor as you use to create your Smartling account.

The mongodb-translators user group exists to facilitate collaboration between translators and the documentation team at large. You can join the Google Group without signing the contributor's agreement.

We currently have the following languages configured:

- Arabic
- Chinese
- Czech
- French
- German
- Hungarian
- Indonesian
- Italian
- Japanese
- Korean
- Lithuanian
- Polish
- Portuguese

- Romanian
- Russian
- Spanish
- Turkish
- Ukrainian

If you would like to initiate a translation project to an additional language, please report this issue using the "*Report a Problem*" link above or by posting to the mongodb-translators list.

Currently the translation project only publishes rendered translation. While the translation effort is currently focused on the web site we are evaluating how to retrieve the translated phrases for use in other media.

See also:

- *Contribute to the Documentation* (page 1239)
- *MongoDB Documentation Style and Conventions* (page 1240)
- *MongoDB Documentation Organization* (page 1248)
- *MongoDB Documentation Practices and Processes* (page 1245)
- *Build and Deploy the MongoDB Documentation* (page 1249)

The entire documentation source for this manual is available in the mongodb/docs repository, which is one of the MongoDB project repositories on GitHub.

To contribute to the documentation, you can open a GitHub account, fork the mongodb/docs repository, make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the MongoDB/10gen Contributor Agreement.

You can clone the repository by issuing the following command at your system shell:

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

68.2 About the Documentation Process

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.

For more information on the MongoDB documentation process, see:

68.2.1 MongoDB Documentation Style and Conventions

This document provides an overview of the style for the MongoDB documentation stored in this repository. The overarching goal of this style guide is to provide an accessible base style to ensure that our documentation is easy to read, simple to use, and straightforward to maintain.

For information regarding the MongoDB Manual organization, see *MongoDB Documentation Organization* (page 1248).

Document History

2011-09-27: Document created with a (very) rough list of style guidelines, conventions, and questions.

2012-01-12: Document revised based on slight shifts in practice, and as part of an effort of making it easier for people outside of the documentation team to contribute to documentation.

2012-03-21: Merged in content from the Jargon, and cleaned up style in light of recent experiences.

2012-08-10: Addition to the "Referencing" section.

2013-02-07: Migrated this document to the manual. Added "map-reduce" terminology convention. Other edits.

Naming Conventions

This section contains guidelines on naming files, sections, documents and other document elements.

- File naming Convention:
  - For Sphinx, all files should have a `.txt` extension.
  - Separate words in file names with hyphens (i.e. `-`.)
  - For most documents, file names should have a terse one or two word name that describes the material covered in the document. Allow the path of the file within the document tree to add some of the required context/categorization. For example it's acceptable to have `http://docs.mongodb.org/manualcore/sharding.rst` and `http://docs.mongodb.org/manualadministration/sharding.rst`.
  - For tutorials, the full title of the document should be in the file name. For example, `http://docs.mongodb.org/manualtutorial/replace-one-configuration-server-in-a-shard-`
- Phrase headlines and titles so that they the content contained within the section so that users can determine what questions the text will answer, and material that it will address without needing them to read the content. This shortens the amount of time that people spend looking for answers, and improvise search/scanning, and possibly "SEO."
- Prefer titles and headers in the form of "Using foo" over "How to Foo."
- When using target references (i.e. `:ref:` references in documents,) use names that include enough context to be intelligible thought all documentations. For example, use "`replica-set-secondary-only-node`" as opposed to "`secondary-only-node`". This is to make the source more usable and easier to maintain.

Style Guide

This includes the local typesetting, English, grammatical, conventions and preferences that all documents in the manual should use. The goal here is to choose good standards, that are clear, and have a stylistic minimalism that does not interfere with or distract from the content. A uniform style will improve user experience, and minimize the effect of a multi-authored document.

Punctuation

- Use the oxford comma.

  Oxford commas are the commas in a list of things (e.g. "something, something else, and another thing") before the conjunction (e.g. "and" or "or.").
- Do not add two spaces after terminal punctuation, such as periods.

- Place commas and periods inside quotation marks.
- Use title case for headings and document titles. Title case capitalizes the first letter of the first, last, and all significant words.

Verbs

Verb tense and mood preferences, with examples:

- **Avoid** the first person. For example do not say, "We will begin the backup process by locking the database," or "I begin the backup process by locking my database instance,"
- **Use** the second person. "If you need to back up your database, start by locking the database first." In practice, however, it's more concise to imply second person using the imperative, as in "Before initiating a backup, lock the database."
- When indicated, use the imperative mood. For example: "Backup your databases often" and "To prevent data loss, back up your databases."
- The future perfect is also useful in some cases. For example, "Creating disk snapshots without locking the database will lead to an inconsistent state."
- Avoid helper verbs, as possible, to increase clarity and concision. For example, attempt to avoid "this does foo" and "this will do foo" when possible. Use "does foo" over "will do foo" in situations where "this foos" is unacceptable.

Referencing

- To refer to future or planned functionality in MongoDB or a driver, *always* link to the Jira case. The Manual's `conf.py` provides an `:issue:` role that links directly to a Jira case (e.g. `:issue:\`SERVER-9001\``).
- For non-object references (i.e. functions, operators, methods, database commands, settings) always reference only the first occurrence of the reference in a section. You should *always* reference objects, except in section headings.
- Structure references with the *why* first; the link second.

  For example, instead of this:

  Use the *Convert a Replica Set to a Replicated Sharded Cluster* (page 524) procedure if you have an existing replica set.

  Type this:

  To deploy a sharded cluster for an existing replica set, see *Convert a Replica Set to a Replicated Sharded Cluster* (page 524).

General Formulations

- Contractions are acceptable insofar as they are necessary to increase readability and flow. Avoid otherwise.
- Make lists grammatically correct.
  - Do not use a period after every item unless the list item completes the unfinished sentence before the list.
  - Use appropriate commas and conjunctions in the list items.
  - Typically begin a bulleted list with an introductory sentence or clause, with a colon or comma.
- The following terms are one word:

  - standalone
  - workflow

- Use "unavailable," "offline," or "unreachable" to refer to a `mongod` instance that cannot be accessed. Do not use the colloquialism "down."
- Always write out units (e.g. "megabytes") rather than using abbreviations (e.g. "MB".)

Structural Formulations

- There should be at least two headings at every nesting level. Within an "h2" block, there should be either: no "h3" blocks, 2 "h3" blocks, or more than 2 "h3" blocks.
- Section headers are in title case (capitalize first, last, and all important words) and should effectively describe the contents of the section. In a single document you should strive to have section titles that are not redundant and grammatically consistent with each other.
- Use paragraphs and paragraph breaks to increase clarity and flow. Avoid burying critical information in the middle of long paragraphs. Err on the side of shorter paragraphs.
- Prefer shorter sentences to longer sentences. Use complex formations only as a last resort, if at all (e.g. compound complex structures that require semi-colons).
- Avoid paragraphs that consist of single sentences as they often represent a sentence that has unintentionally become too complex or incomplete. However, sometimes such paragraphs are useful for emphasis, summary, or introductions.

  As a corollary, most sections should have multiple paragraphs.
- For longer lists and more complex lists, use bulleted items rather than integrating them inline into a sentence.
- Do not expect that the content of any example (inline or blocked,) will be self explanatory. Even when it feels redundant, make sure that the function and use of every example is clearly described.

ReStructured Text and Typesetting

- Place spaces between nested parentheticals and elements in JavaScript examples. For example, prefer `{ [ a, a, a ] }` over `{[a,a,a]}`.
- For underlines associated with headers in RST, use:
  - = for heading level 1 or h1s. Use underlines and overlines for document titles.
  - − for heading level 2 or h2s.
  - ~ for heading level 3 or h3s.
  - ` for heading level 4 or h4s.
- Use hyphens (−) to indicate items of an ordered list.
- Place footnotes and other references, if you use them, at the end of a section rather than the end of a file.

  Use the footnote format that includes automatic numbering and a target name for ease of use. For instance a footnote tag may look like: `[#note]_` with the corresponding directive holding the body of the footnote that resembles the following: `.. [#note]`.

  Do **not** include `.. code-block:: [language]` in footnotes.

- As it makes sense, use the `.. code-block:: [language]` form to insert literal blocks into the text. While the double colon, `::`, is functional, the `.. code-block:: [language]` form makes the source easier to read and understand.
- For all mentions of referenced types (i.e. commands, operators, expressions, functions, statuses, etc.) use the reference types to ensure uniform formatting and cross-referencing.

Jargon and Common Terms

Database Systems and Processes

- To indicate the entire database system, use "MongoDB," not mongo or Mongo.
- To indicate the database process or a server instance, use `mongod` or `mongos`. Refer to these as "processes" or "instances." Reserve "database" for referring to a database structure, i.e., the structure that holds collections and refers to a group of files on disk.

Distributed System Terms

- Refer to partitioned systems as "sharded clusters." Do not use shard clusters or sharded systems.
- Refer to configurations that run with replication as "replica sets" (or "master/slave deployments") rather than "clusters" or other variants.

Data Structure Terms

- "document" refers to "rows" or "records" in a MongoDB database. Potential confusion with "JSON Documents."

  Do not refer to documents as "objects," because drivers (and MongoDB) do not preserve the order of fields when fetching data. If the order of objects matter, use an array.
- "field" refers to a "key" or "identifier" of data within a MongoDB document.
- "value" refers to the contents of a "field".
- "sub-document" describes a nested document.

Other Terms

- Use `example.net` (and `.org` or `.com` if needed) for all examples and samples.
- Hyphenate "map-reduce" in order to avoid ambiguous reference to the command name. Do not camel-case.

Notes on Specific Features

- Geo-Location
  1. While MongoDB *is capable* of storing coordinates in sub-documents, in practice, users should only store coordinates in arrays. (See: DOCS-41.)

68.2.2 MongoDB Documentation Practices and Processes

This document provides an overview of the practices and processes.

Contents

Commits

When relevant, include a Jira case identifier in a commit message. Reference documentation cases when applicable, but feel free to reference other cases from jira.mongodb.org.

Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

For the sake of consistency, remove trailing whitespaces in the source file.

"Hard wrap" files to between 72 and 80 characters per-line.

Standards and Practices

- At least two people should vet all non-trivial changes to the documentation before publication. One of the reviewers should have significant technical experience with the material covered in the documentation.
- All development and editorial work should transpire on GitHub branches or forks that editors can then merge into the publication branches.

Collaboration

To propose a change to the documentation, do either of the following:

- Open a ticket in the documentation project proposing the change. Someone on the documentation team will make the change and be in contact with you so that you can review the change.
- Using GitHub, fork the mongodb/docs repository, commit your changes, and issue a pull request. Someone on the documentation team will review and incorporate your change into the documentation.

Builds

Building the documentation is useful because Sphinx and docutils can catch numerous errors in the format and syntax of the documentation. Additionally, having access to an example documentation as it *will* appear to the users is useful for providing more effective basis for the review process. Besides Sphinx, Pygments, and Python-Docutils, the documentation repository contains all requirements for building the documentation resource.

Talk to someone on the documentation team if you are having problems running builds yourself.

Publication

The makefile for this repository contains targets that automate the publication process. Use `make html` to publish a test build of the documentation in the `build/` directory of your repository. Use `make publish` to build the full contents of the manual from the current branch in the `../public-docs/` directory relative the docs repository.

Other targets include:

- `man` - builds UNIX Manual pages for all Mongodb utilities.
- `push` - builds and deploys the contents of the `../public-docs/`.
- `pdfs` - builds a PDF version of the manual (requires LaTeX dependencies.)

Branches

This section provides an overview of the git branches in the MongoDB documentation repository and their use.

At the present time, future work transpires in the `master`, with the main publication being `current`. As the documentation stabilizes, the documentation team will begin to maintain branches of the documentation for specific MongoDB releases.

Migration from Legacy Documentation

The MongoDB.org Wiki contains a wealth of information. As the transition to the Manual (i.e. this project and resource) continues, it's *critical* that no information disappears or goes missing. The following process outlines *how* to migrate a wiki page to the manual:

1. Read the relevant sections of the Manual, and see what the new documentation has to offer on a specific topic.

   In this process you should follow cross references and gain an understanding of both the underlying information and how the parts of the new content relates its constituent parts.

2. Read the wiki page you wish to redirect, and take note of all of the factual assertions, examples presented by the wiki page.

3. Test the factual assertions of the wiki page to the greatest extent possible. Ensure that example output is accurate. In the case of commands and reference material, make sure that documented options are accurate.

4. Make corrections to the manual page or pages to reflect any missing pieces of information.

   The target of the redirect need *not* contain every piece of information on the wiki page, **if** the manual as a whole does, and relevant section(s) with the information from the wiki page are accessible from the target of the redirection.

5. As necessary, get these changes reviewed by another writer and/or someone familiar with the area of the information in question.

   At this point, update the relevant Jira case with the target that you've chosen for the redirect, and make the ticket unassigned.

6. When someone has reviewed the changes and published those changes to Manual, you, or preferably someone else on the team, should make a final pass at both pages with fresh eyes and then make the redirect.

   Steps 1-5 should ensure that no information is lost in the migration, and that the final review in step 6 should be trivial to complete.

Review Process

Types of Review

The content in the Manual undergoes many types of review, including the following:

Initial Technical Review Review by an engineer familiar with MongoDB and the topic area of the documentation. This review focuses on technical content, and correctness of the procedures and facts presented, but can improve any aspect of the documentation that may still be lacking. When both the initial technical review and the content review are complete, the piece may be "published."

Content Review Textual review by another writer to ensure stylistic consistency with the rest of the manual. Depending on the content, this may precede or follow the initial technical review. When both the initial technical review and the content review are complete, the piece may be "published."

Consistency Review This occurs post-publication and is content focused. The goals of consistency reviews are to increase the internal consistency of the documentation as a whole. Insert relevant cross-references, update the style as needed, and provide background fact-checking.

When possible, consistency reviews should be as systematic as possible and we should avoid encouraging stylistic and information drift by editing only small sections at a time.

Subsequent Technical Review If the documentation needs to be updated following a change in functionality of the server or following the resolution of a user issue, changes may be significant enough to warrant additional technical review. These reviews follow the same form as the "initial technical review," but is often less involved and covers a smaller area.

Review Methods

If you're not a usual contributor to the documentation and would like to review something, you can submit reviews in any of the following methods:

- If you're reviewing an open pull request in GitHub, the best way to comment is on the "overview diff," which you can find by clicking on the "diff" button in the upper left portion of the screen. You can also use the following URL to reach this interface:

  ```
  https://github.com/mongodb/docs/pull/[pull-request-id]/files
  ```

  Replace `[pull-request-id]` with the identifier of the pull request. Make all comments inline, using GitHub's comment system.

  You may also provide comments directly on commits, or on the pull request itself but these commit-comments are archived in less coherent ways and generate less useful emails, while comments on the pull request lead to less specific changes to the document.

- Leave feedback on Jira cases in the DOCS project. These are better for more general changes that aren't necessarily tied to a specific line, or affect multiple files.

- Create a fork of the repository in your GitHub account, make any required changes and then create a pull request with your changes.

  If you insert lines that begin with any of the following annotations:

  ```
  .. TODO:
  TODO:
  .. TODO
  TODO
  ```

  followed by your comments, it will be easier for the original writer to locate your comments. The two dots `..` format is a comment in reStructured Text, which will hide your comments from Sphinx and publication if you're worried about that.

  This format is often easier for reviewers with larger portions of content to review.

68.2.3 MongoDB Documentation Organization

This document provides an overview of the global organization of the documentation resource. Refer to the notes below if you are having trouble understanding the reasoning behind a file's current location, or if you want to add new documentation but aren't sure how to integrate it into the existing resource.

If you have questions, don't hesitate to open a ticket in the Documentation Jira Project or contact the documentation team.

Global Organization

Indexes and Experience

The documentation project has two "index files": `http://docs.mongodb.org/manualcontents.txt` and `http://docs.mongodb.org/manualindex.txt`. The "contents" file provides the documentation's tree structure, which Sphinx uses to create the left-pane navigational structure, to power the "Next" and "Previous" page functionality, and to provide all overarching outlines of the resource. The "index" file is not included in the "contents" file (and thus builds will produce a warning here) and is the page that users first land on when visiting the resource.

Having separate "contents" and "index" files provides a bit more flexibility with the organization of the resource while also making it possible to customize the primary user experience.

Additionally, in the top level of the `source/` directory, there are a number of "topical" index or outline files. These (like the "index" and "contents" files) use the `.. toctree::` directive to provide organization within the documentation. The subject-specific landing pages indexes combine to create the index in the contents file.

Topical Indexes and Meta Organization

Because the documentation on any given subject exists in a number of different locations across the resource the "topical" indexes provide the real structure and organization to the resource. This organization makes it possible to provide great flexibility while still maintaining a reasonable organization of files and URLs for the documentation. Consider the following example:

> Given that topic such as "replication," has material regarding the administration of replica sets, as well as reference material, an overview of the functionality, and operational tutorials, it makes more sense to include a few locations for documents, and use the meta documents to provide the topic-level organization.

Current landing pages include:

- administration

- applications
- crud
- faq
- mongo
- reference
- replication
- security
- sharding

Additional topical indexes are forthcoming.

The Top Level Folders

The documentation has a number of top-level folders, that hold all of the content of the resource. Consider the following list and explanations below:

- "administration" - contains all of the operational and architectural information that systems and database administrators need to know in order to run MongoDB. Topics include: monitoring, replica sets, shard clusters, deployment architectures, and configuration.
- "applications" - contains information about application development and use. While most documentation regarding application development is within the purview of the driver documentation, there are some larger topics regarding the use of these features that deserve some coverage in this context. Topics include: drivers, schema design, optimization, replication, and sharding.
  - "applications/use-cases" - contains use cases that detail how MongoDB can support various kinds uses and application designs, including in depth instructions and examples.
- "core" - contains overviews and introduction to the core features, functionality, and concepts of MongoDB. Topics include: replication, sharding, capped collections, journaling/durability, aggregation.
- "reference" - contains references and indexes of shell functions, database commands, status outputs, as well as manual pages for all of the programs come with MongoDB (e.g. `mongostat` and `mongodump`.)
- "tutorial" - contains operational guides and tutorials that lead users through common tasks (administrative and conceptual) with MongoDB. This includes programming patterns and operational guides.
- "faq" - contains all the frequently asked questions related to MongoDB, in a collection of topical files.

68.2.4 Build and Deploy the MongoDB Documentation

This document contains more direct instructions for building the MongoDB documentation.

Getting Started

Install Dependencies

The MongoDB Documentation project depends on the following tools:

- GNU Make
- GNU Tar

- Python
- Git
- Sphinx (documentation management toolchain)
- Pygments (syntax highlighting)
- PyYAML (for the generated tables)
- Droopy (Python package for static text analysis)
- Fabric (Python package for scripting and orchestration)
- Inkscape (Image generation.)
- python-argparse (For Python 2.6.)
- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Common Utilities (rsync, tar, gzip, sed)

OS X Install Sphinx, Docutils, and their dependencies with `easy_install` the following command:

```
easy_install Sphinx Jinja2 Pygments docutils  PyYAML droopy fabric
```

Feel free to use `pip` rather than `easy_install` to install python packages.

To generate the images used in the documentation, download and install Inkscape.

Optional

To generate PDFs for the full production build, install a TeX distribution (for building the PDF.) If you do not have a LaTeX installation, use MacTeX. This is **only** required to build PDFs.

Arch Linux Install packages from the system repositories with the following command:

```
pacman -S python2-sphinx python2-yaml inkscape python2-pip
```

Then install the following Python packages:

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
pacman -S texlive-bin texlive-core texlive-latexextra
```

Debian/Ubuntu Install the required system packages with the following command:

```
apt-get install python-sphinx python-yaml python-argparse inkscape python-pip
```

Then install the following Python packages:

```
pip install droopy fabric
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
apt-get install texlive-latex-recommended texlive-latex-recommended
```

Setup and Configuration

Clone the repository:

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

Then run the `bootstrap.py` script in the `docs/` repository, to configure the build dependencies:

```
python bootstrap.py
```

This downloads and configures the mongodb/docs-tools repository, which contains the authoritative build system shared between branches of the MongoDB Manual and other MongoDB documentation projects.

You can run `bootstrap.py` regularly to update build system.

Building the Documentation

The MongoDB documentation build system is entirely accessible via `make` targets. For example, to build an HTML version of the documentation issue the following command:

```
make html
```

You can find the build output in `build/<branch>/html`, where `<branch>` is the name of the current branch.

In addition to the `html` target, the build system provides the following targets:

publish Builds and integrates all output for the production build. Build output is in `build/public/<branch>/`. When you run `publish` in the `master`, the build will generate some output in `build/public/`.

push; stage Uploads the production build to the production or staging web servers. Depends on `publish`. Requires access production or staging environment.

push-all; stage-all Uploads the entire content of `build/public/` to the web servers. Depends on `publish`. Not used in common practice.

push-with-delete; stage-with-delete Modifies the action of `push` and `stage` to remove remote file that don't exist in the local build. Use with caution.

generated-source; api; tables; toc; images; installation-guides The build system generates a number of different source files used as inputs to the Sphinx build system. `generated-source` is a meta-target that builds `api`, `tables`, `toc`, `images`, and `installation-guides`.

html; latex; dirhtml; epub; texinfo; man; json These are standard targets derived from the default Sphinx Makefile, with adjusted dependencies. Additionally, for all of these targets you can append `-nitpick` to increase Sphinx's verbosity, or `-clean` to remove all Sphinx build artifacts.

manual-pdfs; manual-info; json-output; The build system does some additional post-processing on some Sphinx output. These targets build the Sphinx dependency.

Build Mechanics and Tools

Internally the build system has a number of components and processes. See the docs-tools README for more information on the internals. This section documents a few of these components from a very high level and lists useful operations for contributors to the documentation.

Fabric

Fabric is an orchestration and scripting package for Python. The documentation uses Fabric to handle the deployment of the build products to the web servers and also unifies a number of independent build operations. Fabric commands have the following form:

```
fab <module>.<task>[:<argument>]
```

The `<argument>` is optional in most cases. Additionally some tasks are available at the root level, without a module. To see a full list of fabric tasks, use the following command:

```
fab -l
```

You can chain fabric tasks on a single command line, although this doesn't always make sense.

Important fabric tasks include:

tools.bootstrap Runs the `bootstrap.py` script. Useful for re-initializing the repository without needing to be in root of the repository.

tools.dev; tools.reset `tools.dev` switches the `origin` remote of the `docs-tools` checkout in `build` directory, to `../docs-tools` to facilitate build system testing and development. `tools.reset` resets the `origin` remote for normal operation.

stats.report:<filename> Returns, a collection of readability statistics. Specify file names relative to `source/` tree. Depends on the build output of the `json-output` Make target.

make Provides a thin wrapper around Make calls. Allows you to start make builds from different locations in the project repository.

process.refresh_dependencies Updates the time stamp of `.txt` source files with changed include files, to facilitate Sphinx's incremental rebuild process.

process.json_output Process the `.fjson` files produced by Sphinx's `json` builder, and the `json` Make target. Unlike `json-output`.

Buildcloth

Buildcloth is a meta-build tool, used to generate Makefiles programatically. This makes the build system easier to maintain, and makes it easier to use the same fundamental code to generate various branches of the Manual as well as related documentation projects. See makecloth/ in the docs-tools repository for the relevant code.

Running `make` with no arguments will regenerate these parts of the build system automatically.

Rstcloth

Rstcloth is a library for generating reStructuredText programatically. This makes it possible to generate content for the documentation, such as tables, tables of contents, and API reference material programatically and transparently. See rstcloth/ in the docs-tools repository for the relevant code.

If you have any questions, please feel free to open a Jira Case.

Symbols

A

B

C

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W