MongoDB Documentation

Release 2.6.11

MongoDB, Inc.

February 12, 2016

Contents

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

[1] http://docs.mongodb.org/v2.6/MongoDB-reference-manual.pdf

CHAPTER 1

Introduction to MongoDB

On this page

Welcome to MongoDB. This document provides a brief introduction to MongoDB and some key concepts. See the *installation guides* (page 5) for information on downloading and installing MongoDB.

1.1 What is MongoDB

MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling.

1.1.1 Document Database

A record in MongoDB is a document, which is a data structure composed of field and value pairs. MongoDB documents are similar to JSON objects. The values of fields may include other documents, arrays, and arrays of documents.

The advantages of using documents are:

- Documents (i.e. objects) correspond to native data types in many programming languages.
- Embedded documents and arrays reduce need for expensive joins.
- Dynamic schema supports fluent polymorphism.

1.1.2 Key Features

High Performance

MongoDB provides high performance data persistence. In particular,

- Support for embedded data models reduces I/O activity on database system.
- Indexes support faster queries and can include keys from embedded documents and arrays.

High Availability

To provide high availability, MongoDB's replication facility, called replica sets, provide:

- *automatic* failover.
- data redundancy.

A *replica set* (page 563) is a group of MongoDB servers that maintain the same data set, providing redundancy and increasing data availability.

Automatic Scaling

MongoDB provides horizontal scalability as part of its *core* functionality.

- Automatic *sharding* (page 675) distributes data across a cluster of machines.
- Replica sets can provide eventually-consistent reads for low-latency high throughput deployments.

CHAPTER 2

Install MongoDB

On this page

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

2.1 Installation Guides

See the *Release Notes* (page 805) for information about specific releases of MongoDB.

Install on Linux **(page 6)** Documentations for installing the official MongoDB distribution on Linux-based systems.

- ***Install on Red Hat*** **(page 6)** Install MongoDB on Red Hat Enterprise and related Linux systems using `.rpm` packages.
- ***Install on Ubuntu*** **(page 10)** Install MongoDB on Ubuntu Linux systems using `.deb` packages.
- ***Install on Debian*** **(page 13)** Install MongoDB on Debian systems using `.deb` packages.
- ***Install on Other Linux Systems*** **(page 16)** Install the official build of MongoDB on other Linux systems from MongoDB archives.

Install on OS X **(page 19)** Install the official build of MongoDB on OS X systems from Homebrew packages or from MongoDB archives.

Install on Windows **(page 21)** Install MongoDB on Windows systems and optionally start MongoDB as a Windows service.

Install MongoDB Enterprise **(page 27)** MongoDB Enterprise is available for MongoDB Enterprise subscribers and includes several additional features including support for SNMP monitoring, LDAP authentication, Kerberos authentication, and System Event Auditing.

- ***Install MongoDB Enterprise on Red Hat*** **(page 28)** Install the MongoDB Enterprise build and required dependencies on Red Hat Enterprise or CentOS Systems using packages.
- ***Install MongoDB Enterprise on Ubuntu*** **(page 32)** Install the MongoDB Enterprise build and required dependencies on Ubuntu Linux Systems using packages.
- ***Install MongoDB Enterprise on Amazon AMI*** **(page 42)** Install the MongoDB Enterprise build and required dependencies on Amazon Linux AMI.

Install MongoDB Enterprise on Windows **(page 44)** Install the MongoDB Enterprise build and required dependencies using the `.msi` installer.

2.1.1 Install on Linux

On this page

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

Recommended

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

Install on Red Hat **(page 6)** Install MongoDB on Red Hat Enterprise and related Linux systems using `.rpm` packages.

Install on Ubuntu **(page 10)** Install MongoDB on Ubuntu Linux systems using `.deb` packages.

Install on Debian **(page 13)** Install MongoDB on Debian systems using `.deb` packages.

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

Manual Installation

For Linux systems without supported packages, see the following guide:

Install on Other Linux Systems **(page 16)** Install the official build of MongoDB on other Linux systems from MongoDB archives.

Install MongoDB on Red Hat Enterprise or CentOS Linux

On this page

Overview Use this tutorial to install MongoDB on Red Hat Enterprise Linux CentOS Linux using `.rpm` packages. While some of these distributions include their own MongoDB packages, the official MongoDB packages are generally more up to date.

Packages MongoDB provides packages of the officially supported MongoDB builds in its own repository. This repository provides the MongoDB distribution in the following packages:

- `mongodb-org`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-org-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-org-mongos`

  This package contains the `mongos` daemon.

- `mongodb-org-shell`

  This package contains the `mongo` shell.

- `mongodb-org-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

Control Scripts The `mongodb-org` package includes various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`. These scripts are used to stop, start, and restart daemon processes.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script for use in such environments. See the `mongos` reference for configuration details.

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

The default `/etc/mongod.conf` configuration file supplied by the 2.6 series packages has `bind_ip` set to `127.0.0.1` by default. Modify this setting as needed for your environment before initializing a *replica set*.

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Install MongoDB

Step 1: Configure the package management system (YUM). Create a `/etc/yum.repos.d/mongodb.repo` file to hold the following configuration information for the MongoDB repository:

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

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

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

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

Step 2: Install the MongoDB packages and associated tools. When you install the packages, you choose whether to install the current release or a previous one. This step provides the commands for both.

To install the latest stable version of MongoDB, issue the following command:

```
sudo yum install -y mongodb-org
```

To install a specific release of MongoDB, specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB:

```
sudo yum install -y mongodb-org-2.6.9 mongodb-org-server-2.6.9 mongodb-org-shell-2.6.9 mongodb-org-mo
```

You can specify any available version of MongoDB. However `yum` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin a package, add the following `exclude` directive to your `/etc/yum.conf` file:

```
exclude=mongodb-org,mongodb-org-server,mongodb-org-shell,mongodb-org-mongos,mongodb-org-tools
```

Previous versions of MongoDB packages use different naming conventions. See the 2.4 version of documentation for more information[1].

Run MongoDB

Prerequisites

Configure SELinux

Important: You must configure SELinux to allow MongoDB to start on Red Hat Linux-based systems (Red Hat Enterprise Linux or CentOS Linux).

To configure SELinux, administrators have three options:

Note: All three options require `root` privileges. The first two options each requires a system reboot and may have larger implications for your deployment.

- Disable SELinux entirely by changing the `SELINUX` setting to `disabled` in `/etc/selinux/config`.

  ```
  SELINUX=disabled
  ```

- Set SELinux to `permissive` mode in `/etc/selinux/config` by changing the `SELINUX` setting to `permissive`.

  ```
  SELINUX=permissive
  ```

  Note: You can use `setenforce` to change to permissive mode; this method does not require a reboot but is **not** persistent.

[1] http://docs.mongodb.org/v2.4/tutorial/install-mongodb-on-linux

- Enable access to the relevant ports (e.g. 27017) for SELinux if in `enforcing` mode. See *Default MongoDB Port* (page 424) for more information on MongoDB's default ports. For default settings, this can be accomplished by running

  ```
  semanage port -a -t mongod_port_t -p tcp 27017
  ```

  Warning: On RHEL 7.0, if you change the data path, the *default* SELinux policies will prevent `mongod` from having write access on the new data path if you do not change the security context.

You may alternatively choose not to install the SELinux packages when you are installing your Linux operating system, or choose to remove the relevant packages. This option is the most invasive and is not recommended.

Data Directories and Permissions

Warning: On RHEL 7.0, if you change the data path, the *default* SELinux policies will preve having write access on the new data path if you do not change the security context.

The MongoDB instance stores its data files in `/var/lib/mongo` and its log files in `/var/log/mongodb` by default, and runs using the `mongod` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. You can start the `mongod` process by issuing the following command:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully You can verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

You can optionally ensure that MongoDB will start following a system reboot by issuing the following command:

```
sudo chkconfig mongod on
```

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. You can restart the `mongod` process by issuing the following command:

```
sudo service mongod restart
```

You can follow the state of the process for errors or important messages by watching the output in the `/var/log/mongodb/mongod.log` file.

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo yum erase $(rpm -qa | grep mongodb-org)
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongo
```

Install MongoDB on Ubuntu

On this page

Overview Use this tutorial to install MongoDB on Ubuntu Linux systems from `.deb` packages. While Ubuntu includes its own MongoDB packages, the official MongoDB packages are generally more up-to-date.

Packages MongoDB provides packages of the officially supported MongoDB builds in its own repository. This repository provides the MongoDB distribution in the following packages:

- `mongodb-org`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-org-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-org-mongos`

  This package contains the `mongos` daemon.

- `mongodb-org-shell`

  This package contains the `mongo` shell.

- `mongodb-org-tools`

  This package contains the following MongoDB tools: `mongoimport` `bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

Control Scripts The `mongodb-org` package includes various *control scripts*, including the init script `/etc/init.d/mongod`. These scripts are used to stop, start, and restart daemon processes.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script for use in such environments. See the `mongos` reference for configuration details.

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

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

The default `/etc/mongod.conf` configuration file supplied by the 2.6 series packages has `bind_ip` set to `127.0.0.1` by default. Modify this setting as needed for your environment before initializing a *replica set*.

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Install MongoDB

Step 1: Import the public key used by the package management system. The Ubuntu package management tools (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys. Issue the following command to import the MongoDB public GPG Key[2]:

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

Step 2: Create a list file for MongoDB. Create the `/etc/apt/sources.list.d/mongodb.list` list file using the following command:

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

Step 3: Reload local package database. Issue the following command to reload the local package database:

```
sudo apt-get update
```

Step 4: Install the MongoDB packages. You can install either the latest stable version of MongoDB or a specific version of MongoDB.

[2] http://docs.mongodb.org/10gen-gpg-key.asc

Install the latest stable version of MongoDB. Issue the following command:

```
sudo apt-get install -y mongodb-org
```

Install a specific release of MongoDB. Specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB:

```
sudo apt-get install -y mongodb-org=2.6.9 mongodb-org-server=2.6.9 mongodb-org-shell=2.6.9 mongodb-or
```

Pin a specific version of MongoDB. Although you can specify any available version of MongoDB, `apt-get` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin the version of MongoDB at the currently installed version, issue the following command sequence:

```
echo "mongodb-org hold" | sudo dpkg --set-selections
echo "mongodb-org-server hold" | sudo dpkg --set-selections
echo "mongodb-org-shell hold" | sudo dpkg --set-selections
echo "mongodb-org-mongos hold" | sudo dpkg --set-selections
echo "mongodb-org-tools hold" | sudo dpkg --set-selections
```

Previous versions of MongoDB packages use different naming conventions. See the 2.4 version of documentation for more information[3].

Run MongoDB The MongoDB instance stores its data files in `/var/lib/mongodb` and its log files in `/var/log/mongodb` by default, and runs using the `mongodb` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongodb` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. Issue the following command to start `mongod`:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully Verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. Issue the following command to restart `mongod`:

```
sudo service mongod restart
```

[3] http://docs.mongodb.org/v2.4/tutorial/install-mongodb-on-ubuntu

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo apt-get purge mongodb-org*
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongodb
```

Install MongoDB on Debian

On this page

Overview Use this tutorial to install MongoDB from `.deb` packages on Debian 7. While Debian includes its own MongoDB packages, the official MongoDB packages are more up to date.

Packages MongoDB provides packages of the officially supported MongoDB builds in its own repository. This repository provides the MongoDB distribution in the following packages:

- `mongodb-org`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-org-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-org-mongos`

  This package contains the `mongos` daemon.

- `mongodb-org-shell`

  This package contains the `mongo` shell.

- `mongodb-org-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

Control Scripts The `mongodb-org` package includes various *control scripts*, including the init script `/etc/init.d/mongod`. These scripts are used to stop, start, and restart daemon processes.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script for use in such environments. See the `mongos` reference for configuration details.

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

You cannot install this package concurrently with the `mongodb`, `mongodb-server`, or `mongodb-clients` packages included in Debian 7.

The default `/etc/mongod.conf` configuration file supplied by the 2.6 series packages has `bind_ip` set to `127.0.0.1` by default. Modify this setting as needed for your environment before initializing a *replica set*.

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Install MongoDB The Debian package management tools (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys.

Step 1: Import the public key used by the package management system. Issue the following command to add the MongoDB public GPG Key[4] to the system key ring.

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

Step 2: Create a `/etc/apt/sources.list.d/mongodb.list` file for MongoDB. Create the list file using the following command:

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

Step 3: Reload local package database. Issue the following command to reload the local package database:

```
sudo apt-get update
```

[4] http://docs.mongodb.org/10gen-gpg-key.asc

Step 4: Install the MongoDB packages. You can install either the latest stable version of MongoDB or a specific version of MongoDB.

Install the latest stable version of MongoDB. Issue the following command:

```
sudo apt-get install -y mongodb-org
```

Install a specific release of MongoDB. Specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB:

```
sudo apt-get install -y mongodb-org=2.6.9 mongodb-org-server=2.6.9 mongodb-org-shell=2.6.9 mongodb-or
```

Pin a specific version of MongoDB. Although you can specify any available version of MongoDB, `apt-get` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin the version of MongoDB at the currently installed version, issue the following command sequence:

```
echo "mongodb-org hold" | sudo dpkg --set-selections
echo "mongodb-org-server hold" | sudo dpkg --set-selections
echo "mongodb-org-shell hold" | sudo dpkg --set-selections
echo "mongodb-org-mongos hold" | sudo dpkg --set-selections
echo "mongodb-org-tools hold" | sudo dpkg --set-selections
```

Previous versions of MongoDB packages use different naming conventions. See the 2.4 version of documentation for more information[5].

Run MongoDB The MongoDB instance stores its data files in `/var/lib/mongodb` and its log files in `/var/log/mongodb` by default, and runs using the `mongodb` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongodb` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. Issue the following command to start `mongod`:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully Verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

[5] http://docs.mongodb.org/v2.4/tutorial/install-mongodb-on-ubuntu

Step 4: Restart MongoDB. Issue the following command to restart `mongod`:

```
sudo service mongod restart
```

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

Warning: This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo apt-get purge mongodb-org*
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongodb
```

Install MongoDB on Linux Systems

On this page

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

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

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

Install for 64-bit Linux

Step 1: Download the binary files for the desired release of MongoDB. Download the binaries from `https://www.mongodb.org/downloads`.

For example, to download the latest release through the shell, issue the following:

```
curl -O http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.6.11.tgz
```

Step 2: Extract the files from the downloaded archive. For example, from a system shell, you can extract through the `tar` command:

```
tar -zxvf mongodb-linux-x86_64-2.6.11.tgz
```

Step 3: Copy the extracted archive to the target directory. Copy the extracted folder to the location from which MongoDB will run.

```
mkdir -p mongodb
cp -R -n mongodb-linux-x86_64-2.6.11/ mongodb
```

Step 4: Ensure the location of the binaries is in the `PATH` variable. The MongoDB binaries are in the `bin/` directory of the archive. To ensure that the binaries are in your `PATH`, you can modify your `PATH`.

For example, you can add the following line to your shell's `rc` file (e.g. `~/.bashrc`):

```
export PATH=<mongodb-install-directory>/bin:$PATH
```

Replace `<mongodb-install-directory>` with the path to the extracted MongoDB archive.

Install for 32-bit Linux

Step 1: Download the binary files for the desired release of MongoDB. Download the binaries from `https://www.mongodb.org/downloads`.

For example, to download the latest release through the shell, issue the following:

```
curl -O http://downloads.mongodb.org/linux/mongodb-linux-i686-2.6.11.tgz
```

Step 2: Extract the files from the downloaded archive. For example, from a system shell, you can extract through the `tar` command:

```
tar -zxvf mongodb-linux-i686-2.6.11.tgz
```

Step 3: Copy the extracted archive to the target directory. Copy the extracted folder to the location from which MongoDB will run.

```
mkdir -p mongodb
cp -R -n mongodb-linux-i686-2.6.11/ mongodb
```

Step 4: Ensure the location of the binaries is in the `PATH` variable. The MongoDB binaries are in the `bin/` directory of the archive. To ensure that the binaries are in your `PATH`, you can modify your `PATH`.

For example, you can add the following line to your shell's `rc` file (e.g. `~/.bashrc`):

```
export PATH=<mongodb-install-directory>/bin:$PATH
```

Replace `<mongodb-install-directory>` with the path to the extracted MongoDB archive.

Run MongoDB

Step 1: Create the data directory. Before you start MongoDB for the first time, create the directory to which the `mongod` process will write data. By default, the `mongod` process uses the `/data/db` directory. If you create a directory other than this one, you must specify that directory in the `dbpath` option when starting the `mongod` process later in this procedure.

The following example command creates the default `/data/db` directory:

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

Step 2: Set permissions for the data directory. Before running `mongod` for the first time, ensure that the user account running `mongod` has read and write permissions for the directory.

Step 3: Run MongoDB. To run MongoDB, run the `mongod` process at the system prompt. If necessary, specify the path of the `mongod` or the data directory. See the following examples.

Run without specifying paths If your system `PATH` variable includes the location of the `mongod` binary and if you use the default data directory (i.e., `/data/db`), simply enter `mongod` at the system prompt:

```
mongod
```

Specify the path of the `mongod` If your `PATH` does not include the location of the `mongod` binary, enter the full path to the `mongod` binary at the system prompt:

```
<path to binary>/mongod
```

Specify the path of the data directory If you do not use the default data directory (i.e., `/data/db`), specify the path to the data directory using the *`--dbpath`* option:

```
mongod --dbpath <path to data directory>
```

Step 4: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

2.1.2 Install MongoDB on OS X

On this page

Overview

Use this tutorial to install MongoDB on OS X systems.

Platform Support

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

MongoDB is available through the popular OS X package manager Homebrew[6] or through the MongoDB Download site[7].

Install MongoDB

You can install MongoDB with Homebrew[8] or manually. This section describes both.

Install MongoDB with Homebrew

Homebrew[9] installs binary packages based on published "formulae." This section describes how to update `brew` to the latest packages and install MongoDB. Homebrew requires some initial setup and configuration, which is beyond the scope of this document.

Step 1: Update Homebrew's package database.

In a system shell, issue the following command:

```
brew update
```

Step 2: Install MongoDB.

You can install MongoDB via `brew` with several different options. Use one of the following operations:

Install the MongoDB Binaries To install the MongoDB binaries, issue the following command in a system shell:

```
brew install mongodb
```

[6] http://brew.sh/
[7] http://www.mongodb.org/downloads
[8] http://brew.sh/
[9] http://brew.sh/

Build MongoDB from Source with TLS/SSL Support To build MongoDB from the source files and include TLS/SSL support, issue the following from a system shell:

```
brew install mongodb --with-openssl
```

Install the Latest Development Release of MongoDB To install the latest development release for use in testing and development, issue the following command in a system shell:

```
brew install mongodb --devel
```

Install MongoDB Manually

Only install MongoDB using this procedure if you cannot use *homebrew* (page 19).

Step 1: Download the binary files for the desired release of MongoDB.

Download the binaries from `https://www.mongodb.org/downloads`.

For example, to download the latest release through the shell, issue the following:

```
curl -O http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.11.tgz
```

Step 2: Extract the files from the downloaded archive.

For example, from a system shell, you can extract through the `tar` command:

```
tar -zxvf mongodb-osx-x86_64-2.6.11.tgz
```

Step 3: Copy the extracted archive to the target directory.

Copy the extracted folder to the location from which MongoDB will run.

```
mkdir -p mongodb
cp -R -n mongodb-osx-x86_64-2.6.11/ mongodb
```

Step 4: Ensure the location of the binaries is in the `PATH` variable.

The MongoDB binaries are in the `bin/` directory of the archive. To ensure that the binaries are in your `PATH`, you can modify your `PATH`.

For example, you can add the following line to your shell's `rc` file (e.g. `~/.bashrc`):

```
export PATH=<mongodb-install-directory>/bin:$PATH
```

Replace `<mongodb-install-directory>` with the path to the extracted MongoDB archive.

Run MongoDB

Step 1: Create the data directory.

Before you start MongoDB for the first time, create the directory to which the `mongod` process will write data. By default, the `mongod` process uses the `/data/db` directory. If you create a directory other than this one, you must specify that directory in the `dbpath` option when starting the `mongod` process later in this procedure.

The following example command creates the default `/data/db` directory:

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

Step 2: Set permissions for the data directory.

Before running `mongod` for the first time, ensure that the user account running `mongod` has read and write permissions for the directory.

Step 3: Run MongoDB.

To run MongoDB, run the `mongod` process at the system prompt. If necessary, specify the path of the `mongod` or the data directory. See the following examples.

Run without specifying paths If your system `PATH` variable includes the location of the `mongod` binary and if you use the default data directory (i.e., `/data/db`), simply enter `mongod` at the system prompt:

```
mongod
```

Specify the path of the `mongod` If your `PATH` does not include the location of the `mongod` binary, enter the full path to the `mongod` binary at the system prompt:

```
<path to binary>/mongod
```

Specify the path of the data directory If you do not use the default data directory (i.e., `/data/db`), specify the path to the data directory using the *`--dbpath`* option:

```
mongod --dbpath <path to data directory>
```

Step 4: Begin using MongoDB.

To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

2.1.3 Install MongoDB on Windows

On this page

Overview

Use this tutorial to install MongoDB on a Windows systems.

Platform Support

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

Important: If you are running any edition of Windows Server 2008 R2 or Windows 7, please install a hotfix to resolve an issue with memory mapped files on Windows[10].

Requirements

On Windows MongoDB requires Windows Server 2008 R2, Windows Vista, or later. The `.msi` installer includes all other software dependencies and will automatically upgrade any older version of MongoDB installed using an `.msi` file.

Get MongoDB

Step 1: Determine which MongoDB build you need.

There are three builds of MongoDB for Windows:

MongoDB for Windows 64-bit runs only 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 32-bit runs on any 32-bit version of Windows newer than Windows Vista. 32-bit versions of MongoDB are only intended for older systems and for use in testing and development systems. 32-bit versions of MongoDB only support databases smaller than 2GB.

MongoDB for Windows 64-bit Legacy runs on Windows Vista, Windows Server 2003, and Windows Server 2008 and does not include recent performance enhancements.

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

```
wmic os get osarchitecture
```

[10] http://support.microsoft.com/kb/2731284

Step 2: Download MongoDB for Windows.

Download the latest production release of MongoDB from the MongoDB downloads page[11]. Ensure you download the correct version of MongoDB for your Windows system. The 64-bit versions of MongoDB do not work with 32-bit Windows.

Install MongoDB

Interactive Installation

Step 1: Install MongoDB for Windows.

In Windows Explorer, locate the downloaded MongoDB `.msi` file, which typically is located in the default `Downloads` folder. Double-click the `.msi` file. A set of screens will appear to guide you through the installation process.

You may specify an installation directory if you choose the "Custom" installation option.

Note: These instructions assume that you have installed MongoDB to `C:\mongodb`.

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 folder (e.g. `D:\test\mongodb`).

Unattended Installation

You may install MongoDB unattended on Windows from the command line using `msiexec.exe`.

Step 1: Install MongoDB for Windows.

Open a shell in the directory containing the `.msi` installation binary of your choice and invoke:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="<installation directory>"
```

By default, this method installs the following MongoDB binaries: `mongod.exe`, `mongo.exe`, `mongodump.exe`, `mongorestore.exe`, `mongoimport.exe`, `mongoexport.exe`, `mongostat.exe`, and `mongotop.exe`. You can specify the installation location for the executable by modifying the `<installation directory>` value. To install specific subsets of the binaries, you may specify an `ADDLOCAL` argument:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="<installation directory>" ADDLOCAL=<b
```

The `<binary set(s)>` value is a comma-separated list including one or more of the following:

- `Server` - includes `mongod.exe`
- `Client` - includes `mongo.exe`
- `MonitoringTools` - includes `mongostat.exe` and `mongotop.exe`
- `ImportExportTools` - includes `mongodump.exe`, `mongorestore.exe`, `mongoexport.exe`, and `mongoimport.exe`)
- `MiscellaneousTools` - includes `bsondump.exe`, `mongofiles.exe`, `mongooplog.exe`, and `mongoperf.exe`

[11] http://www.mongodb.org/downloads

For instance, to install *only* the entire set of tools to `C:\mongodb`, invoke:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="C:\mongodb" ADDLOCAL=MonitoringTools,
```

You may also specify `ADDLOCAL=ALL` to install the complete set of binaries, as in the following:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="C:\mongodb" ADDLOCAL=ALL
```

Run MongoDB

Warning: Do not make `mongod.exe` visible on public networks without running in "Secure Mode" with the `auth` setting. MongoDB is designed to be run in trusted environments, and the database does not enable "Secure Mode" by default.

Step 1: Set up the MongoDB environment.

MongoDB requires a *data directory* to store all data. MongoDB's default data directory path is `\data\db`. Create this folder using the following commands from a *Command Prompt*:

```
md \data\db
```

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

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

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

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

You may also specify the `dbpath` in a `configuration file`.

Step 2: Start MongoDB.

To start MongoDB, run `mongod.exe`. For example, from the *Command Prompt*:

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

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

Depending on the security level of your system, Windows may pop up 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 see the *Security Documentation* (page 316).

Step 3: Connect to MongoDB.

To connect to MongoDB through the `mongo.exe` shell, open another *Command Prompt*.

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

If you want to develop applications using .NET, see the documentation of C# and MongoDB[12] for more information.

[12] https://docs.mongodb.org/ecosystem/drivers/csharp

Step 4: Begin using MongoDB.

To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Configure a Windows Service for MongoDB

Step 1: Open an Administrator command prompt.

Windows 7 / Vista / Server 2008 (and R2) Press `Win + R`, then type `cmd`, then press `Ctrl + Shift + Enter`.

Windows 8 Press `Win + X`, then press `A`.

Execute the remaining steps from the Administrator command prompt.

Step 2: Create directories.

Create directories for your database and log files:

```
mkdir c:\data\db
mkdir c:\data\log
```

Step 3: Create a configuration file.

Create a configuration file. The file **must** set `systemLog.path`. Include additional `configuration options` as appropriate.

For example, create a file at `C:\mongodb\mongod.cfg` that specifies both `systemLog.path` and `storage.dbPath`:

```
systemLog:
    destination: file
    path: c:\data\log\mongod.log
storage:
    dbPath: c:\data\db
```

Step 4: Install the MongoDB service.

Important: Run all of the following commands in *Command Prompt* with "Administrative Privileges".

Install the MongoDB service by starting `mongod.exe` with the `--install` option and the `-config` option to specify the previously created configuration file.

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

To use an alternate `dbpath`, specify the path in the configuration file (e.g. `C:\mongodb\mongod.cfg`) or on the command line with the *`--dbpath`* option.

If needed, you can install services for multiple instances of `mongod.exe` or `mongos.exe`. Install each service with a unique *`--serviceName`* and *`--serviceDisplayName`*. Use multiple instances only when sufficient system resources exist and your system design requires it.

Step 5: Start the MongoDB service.

```
net start MongoDB
```

Step 6: Stop or remove the MongoDB service as needed.

To stop the MongoDB service use the following command:

```
net stop MongoDB
```

To remove the MongoDB service use the following command:

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

Manually Create a Windows Service for MongoDB

You can set up the MongoDB server as a *Windows Service* that starts automatically at boot time.

The following procedure assumes you have installed MongoDB using the `.msi` installer with the path `C:\mongodb\`.

If you have installed in an alternative directory, you will need to adjust the paths as appropriate.

Step 1: Open an Administrator command prompt.

Windows 7 / Vista / Server 2008 (and R2) Press `Win + R`, then type `cmd`, then press `Ctrl + Shift + Enter`.

Windows 8 Press `Win + X`, then press `A`.

Execute the remaining steps from the Administrator command prompt.

Step 2: Create directories.

Create directories for your database and log files:

```
mkdir c:\data\db
mkdir c:\data\log
```

Step 3: Create a configuration file.

Create a configuration file. The file **must** set `systemLog.path`. Include additional `configuration options` as appropriate.

For example, create a file at `C:\mongodb\mongod.cfg` that specifies both `systemLog.path` and `storage.dbPath`:

```
systemLog:
    destination: file
    path: c:\data\log\mongod.log
storage:
    dbPath: c:\data\db
```

Step 4: Create the MongoDB service.

Create the MongoDB service.

```
sc.exe create MongoDB binPath= "\"C:\mongodb\mongod.exe\" --service --config= \"C:\mongodb\mongod.cfg
```

`sc.exe` requires a space between "=" and the configuration values (eg "binPath= "), and a "\" to escape double quotes.

If successfully created, the following log message will display:

```
[SC] CreateService SUCCESS
```

Step 5: Start the MongoDB service.

```
net start MongoDB
```

Step 6: Stop or remove the MongoDB service as needed.

To stop the MongoDB service, use the following command:

```
net stop MongoDB
```

To remove the MongoDB service, first stop the service and then run the following command:

```
sc.exe delete MongoDB
```

Additional Resources

- MongoDB for Developers Free Course[13]
- MongoDB for .NET Developers Free Online Course[14]
- MongoDB Architecture Guide[15]

2.1.4 Install MongoDB Enterprise

These documents provide instructions to install MongoDB Enterprise for Linux and Windows Systems.

Install MongoDB Enterprise on Red Hat **(page 28)** Install the MongoDB Enterprise build and required dependencies on Red Hat Enterprise or CentOS Systems using packages.

Install MongoDB Enterprise on Ubuntu **(page 32)** Install the MongoDB Enterprise build and required dependencies on Ubuntu Linux Systems using packages.

[13] https://university.mongodb.com/courses/M101P/about?jmp=docs
[14] https://university.mongodb.com/courses/M101N/about?jmp=docs
[15] https://www.mongodb.com/lp/white-paper/architecture-guide?jmp=docs

Install MongoDB Enterprise on Debian **(page 36)** Install the MongoDB Enterprise build and required dependencies on Debian Linux Systems using packages.

Install MongoDB Enterprise on SUSE **(page 39)** Install the MongoDB Enterprise build and required dependencies on SUSE Enterprise Linux.

Install MongoDB Enterprise on Amazon AMI **(page 42)** Install the MongoDB Enterprise build and required dependencies on Amazon Linux AMI.

Install MongoDB Enterprise on Windows **(page 44)** Install the MongoDB Enterprise build and required dependencies using the `.msi` installer.

Install MongoDB Enterprise on Red Hat Enterprise or CentOS

On this page

Overview

Use this tutorial to install MongoDB Enterprise[16] on Red Hat Enterprise Linux or CentOS Linux from `.rpm` packages.

Packages

MongoDB provides packages of the officially supported MongoDB Enterprise builds in it's own repository. This repository provides the MongoDB Enterprise distribution in the following packages:

- `mongodb-enterprise`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-enterprise-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-enterprise-mongos`

  This package contains the `mongos` daemon.

- `mongodb-enterprise-shell`

  This package contains the `mongo` shell.

- `mongodb-enterprise-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongoimport`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

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

Control Scripts

The `mongodb-enterprise` package includes various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script.

Considerations

MongoDB only provides Enterprise packages for 64-bit builds of Red Hat Enterprise Linux and CentOS Linux versions 5, 6, and 7.

Use the provided distribution packages as described in this page if possible. These packages will automatically install all of MongoDB's dependencies, and are the recommended installation method.

To manually install all dependencies, run the appropriate command for your Red Hat/CentOS version.

Version 5

```
yum install perl cyrus-sasl cyrus-sasl-plain cyrus-sasl-gssapi krb5-libs \
            lm_sensors net-snmp openssl popt rpm-libs tcp_wrappers zlib
```

Version 6

```
yum install cyrus-sasl cyrus-sasl-plain cyrus-sasl-gssapi krb5-libs \
            net-snmp openssl
```

Version 7

```
yum install cyrus-sasl cyrus-sasl-plain cyrus-sasl-gssapi krb5-libs \
            lm_sensors-libs net-snmp-agent-libs net-snmp openssl rpm-libs \
            tcp_wrappers-libs
```

The default `/etc/mongod.conf` configuration file supplied by the 2.6 series packages has `bind_ip` set to `127.0.0.1` by default. Modify this setting as needed for your environment before initializing a *replica set*.

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Install MongoDB Enterprise

When you install the packages for MongoDB Enterprise, you choose whether to install the current release or a previous one. This procedure describes how to do both.

Step 1: Configure repository. Create an `/etc/yum.repos.d/mongodb-enterprise.repo` file so that you can install MongoDB enterprise directly, using `yum`.

Use the following repository file to specify the *latest* stable release of MongoDB enterprise.

```
[mongodb-enterprise]
name=MongoDB Enterprise Repository
baseurl=https://repo.mongodb.com/yum/redhat/$releasever/mongodb-enterprise/stable/$basearch/
gpgcheck=0
enabled=1
```

Use the following repository to install *only* versions of MongoDB for the `2.6` release. If you'd like to install MongoDB Enterprise packages from a particular *release series* (page 908), such as 2.4 or 2.6, you can specify the release series in the repository configuration. For example, to restrict your system to the 2.6 release series, create a `/etc/yum.repos.d/mongodb-enterprise-2.6.repo` file to hold the following configuration information for the MongoDB Enterprise 2.6 repository:

```
[mongodb-enterprise-2.6]
name=MongoDB Enterprise 2.6 Repository
baseurl=https://repo.mongodb.com/yum/redhat/$releasever/mongodb-enterprise/2.6/$basearch/
gpgcheck=0
enabled=1
```

`.repo` files for each release can also be found in the repository itself[17]. Remember that odd-numbered minor release versions (e.g. 2.5) are development versions and are unsuitable for production deployment.

Step 2: Install the MongoDB Enterprise packages and associated tools. You can install either the latest stable version of MongoDB Enterprise or a specific version of MongoDB Enterprise.

To install the latest stable version of MongoDB Enterprise, issue the following command:

```
sudo yum install -y mongodb-enterprise
```

Step 3: Optional: Manage Installed Version

Install a specific release of MongoDB Enterprise. Specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB:

```
sudo yum install -y mongodb-enterprise-2.6.9 mongodb-enterprise-server-2.6.9 mongodb-enterprise-shell
```

Pin a specific version of MongoDB Enterprise. Although you can specify any available version of MongoDB Enterprise, `yum` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin a package, add the following `exclude` directive to your `/etc/yum.conf` file:

```
exclude=mongodb-enterprise,mongodb-enterprise-server,mongodb-enterprise-shell,mongodb-enterprise-mong
```

Previous versions of MongoDB packages use different naming conventions. See the 2.4 version of documentation for more information[18].

Step 4: When the install completes, you can run MongoDB.

Run MongoDB Enterprise

Prerequisites

Configure SELinux

Important: You must configure SELinux to allow MongoDB to start on Red Hat Linux-based systems (Red Hat Enterprise Linux or CentOS Linux).

To configure SELinux, administrators have three options:

[17] https://repo.mongodb.com/yum/redhat/
[18] http://docs.mongodb.org/v2.4/tutorial/install-mongodb-on-linux

Note: All three options require `root` privileges. The first two options each requires a system reboot and may have larger implications for your deployment.

- Disable SELinux entirely by changing the `SELINUX` setting to `disabled` in `/etc/selinux/config`.

  ```
  SELINUX=disabled
  ```

- Set SELinux to `permissive` mode in `/etc/selinux/config` by changing the `SELINUX` setting to `permissive`.

  ```
  SELINUX=permissive
  ```

  Note: You can use `setenforce` to change to permissive mode; this method does not require a reboot but is **not** persistent.

- Enable access to the relevant ports (e.g. 27017) for SELinux if in `enforcing` mode. See *Default MongoDB Port* (page 424) for more information on MongoDB's default ports. For default settings, this can be accomplished by running

  ```
  semanage port -a -t mongod_port_t -p tcp 27017
  ```

  Warning: On RHEL 7.0, if you change the data path, the *default* SELinux policies will prevent `mongod` from having write access on the new data path if you do not change the security context.

You may alternatively choose not to install the SELinux packages when you are installing your Linux operating system, or choose to remove the relevant packages. This option is the most invasive and is not recommended.

Data Directories and Permissions

Warning: On RHEL 7.0, if you change the data path, the *default* SELinux policies will preve having write access on the new data path if you do not change the security context.

The MongoDB instance stores its data files in `/var/lib/mongo` and its log files in `/var/log/mongodb` by default, and runs using the `mongod` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. You can start the `mongod` process by issuing the following command:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully You can verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

You can optionally ensure that MongoDB will start following a system reboot by issuing the following command:

```
sudo chkconfig mongod on
```

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. You can restart the `mongod` process by issuing the following command:

```
sudo service mongod restart
```

You can follow the state of the process for errors or important messages by watching the output in the `/var/log/mongodb/mongod.log` file.

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo yum erase $(rpm -qa | grep mongodb-enterprise)
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongo
```

Install MongoDB Enterprise on Ubuntu

On this page

Overview

Use this tutorial to install MongoDB Enterprise[19] on Ubuntu Linux systems from `.deb` packages.

Packages

MongoDB provides packages of the officially supported MongoDB Enterprise builds in it's own repository. This repository provides the MongoDB Enterprise distribution in the following packages:

- `mongodb-enterprise`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-enterprise-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-enterprise-mongos`

  This package contains the `mongos` daemon.

- `mongodb-enterprise-shell`

  This package contains the `mongo` shell.

- `mongodb-enterprise-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongoimport`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

Control Scripts

The `mongodb-enterprise` package includes various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script.

Considerations

MongoDB only provides Enterprise packages for Ubuntu 12.04 LTS (Precise Pangolin) and 14.04 LTS (Trusty Tahr).

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Use the provided distribution packages as described in this page if possible. These packages will automatically install all of MongoDB's dependencies, and are the recommended installation method.

To manually install all dependencies, run the following command:

```
sudo apt-get install libgssapi-krb5-2 libsasl2-2 libssl1.0.0 libstdc++6 snmp
```

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

Install MongoDB Enterprise

Step 1: Import the public key used by the package management system. The Ubuntu package management tools (i.e. `dpkg` and `apt`) ensure package consistency and authenticity by requiring that distributors sign packages with GPG keys. Issue the following command to import the MongoDB public GPG Key[20]:

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

Step 2: Create a `/etc/apt/sources.list.d/mongodb-enterprise.list` file for MongoDB. Create the list file using the following command:

```
echo "deb http://repo.mongodb.com/apt/ubuntu "$(lsb_release -sc)"/mongodb-enterprise/stable multivers
```

If you'd like to install MongoDB Enterprise packages from a particular *release series* (page 908), such as 2.4 or 2.6, you can specify the release series in the repository configuration. For example, to restrict your system to the 2.6 release series, add the following repository:

```
echo "deb http://repo.mongodb.com/apt/ubuntu "$(lsb_release -sc)"/mongodb-enterprise/2.6 multiverse"
```

Step 3: Reload local package database. Issue the following command to reload the local package database:

```
sudo apt-get update
```

Step 4: Install the MongoDB Enterprise packages. When you install the packages, you choose whether to install the current release or a previous one. This step provides instructions for both.

To install the latest stable version of MongoDB Enterprise, issue the following command:

```
sudo apt-get install mongodb-enterprise
```

To install a specific release of MongoDB Enterprise, specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB Enterprise:

```
sudo apt-get install mongodb-enterprise=2.6.9 mongodb-enterprise-server=2.6.9 mongodb-enterprise-shel
```

You can specify any available version of MongoDB Enterprise. However `apt-get` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin the version of MongoDB Enterprise at the currently installed version, issue the following command sequence:

```
echo "mongodb-enterprise hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-server hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-shell hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-mongos hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-tools hold" | sudo dpkg --set-selections
```

Previous versions of MongoDB Enterprise packages use different naming conventions. See the 2.4 version of documentation[21] for more information.

[20] http://docs.mongodb.org/10gen-gpg-key.asc
[21] http://docs.mongodb.org/v2.4/tutorial/install-mongodb-enterprise

Run MongoDB Enterprise

The MongoDB instance stores its data files in `/var/lib/mongodb` and its log files in `/var/log/mongodb` by default, and runs using the `mongodb` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongodb` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. Issue the following command to start `mongod`:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully Verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. Issue the following command to restart `mongod`:

```
sudo service mongod restart
```

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo apt-get purge mongodb-enterprise*
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongodb
```

Install MongoDB Enterprise on Debian

On this page

Overview

Use this tutorial to install MongoDB Enterprise[22] from `.deb` packages on Debian 7.

Packages

MongoDB provides packages of the officially supported MongoDB Enterprise builds in it's own repository. This repository provides the MongoDB Enterprise distribution in the following packages:

- `mongodb-enterprise`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-enterprise-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-enterprise-mongos`

  This package contains the `mongos` daemon.

- `mongodb-enterprise-shell`

  This package contains the `mongo` shell.

- `mongodb-enterprise-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongoimport`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

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

Control Scripts

The `mongodb-enterprise` package includes various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script.

Considerations

Changed in version 2.6: The package structure and names have changed as of version 2.6. For instructions on installation of an older release, please refer to the documentation for the appropriate version.

Use the provided distribution packages as described in this page if possible. These packages will automatically install all of MongoDB's dependencies, and are the recommended installation method.

To manually install all dependencies, run the following command:

```
sudo apt-get install libgssapi-krb5-2 libsasl2-2 libssl1.0.0 libstdc++6 snmp
```

Install MongoDB Enterprise

Step 1: Import the public key used by the package management system. Issue the following command to add the MongoDB public GPG Key[23] to the system key ring.

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

Step 2: Create a `/etc/apt/sources.list.d/mongodb-enterprise.list` file for MongoDB. Create the list file using the following command:

```
echo "deb http://repo.mongodb.com/apt/debian "$(lsb_release -sc)"/mongodb-enterprise/stable main" | s
```

If you'd like to install MongoDB Enterprise packages from a particular *release series* (page 908), such as 2.6, you can specify the release series in the repository configuration. For example, to restrict your system to the 2.6 release series, add the following repository:

```
echo "deb http://repo.mongodb.com/apt/debian "$(lsb_release -sc)"/mongodb-enterprise/2.6 main" | sudo
```

Step 3: Reload local package database. Issue the following command to reload the local package database:

```
sudo apt-get update
```

Step 4: Install the MongoDB Enterprise packages. When you install the packages, you choose whether to install the current release or a previous one. This step provides instructions for both.

To install the latest stable version of MongoDB Enterprise, issue the following command:

```
sudo apt-get install mongodb-enterprise
```

[23] http://docs.mongodb.org/10gen-gpg-key.asc

To install a specific release of MongoDB Enterprise, specify each component package individually and append the version number to the package name, as in the following example that installs the `2.6.9` release of MongoDB Enterprise:

```
sudo apt-get install mongodb-enterprise=2.6.9 mongodb-enterprise-server=2.6.9 mongodb-enterprise-shel
```

You can specify any available version of MongoDB Enterprise. However `apt-get` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the package. To pin the version of MongoDB Enterprise at the currently installed version, issue the following command sequence:

```
echo "mongodb-enterprise hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-server hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-shell hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-mongos hold" | sudo dpkg --set-selections
echo "mongodb-enterprise-tools hold" | sudo dpkg --set-selections
```

Run MongoDB Enterprise

The MongoDB instance stores its data files in `/var/lib/mongodb` and its log files in `/var/log/mongodb` by default, and runs using the `mongodb` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongodb` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. Issue the following command to start `mongod`:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully Verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. Issue the following command to restart `mongod`:

```
sudo service mongod restart
```

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo apt-get purge mongodb-enterprise*
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongodb
```

Install MongoDB Enterprise on SUSE

On this page

Overview

Use this tutorial to install MongoDB Enterprise[24] on SUSE Linux. MongoDB Enterprise is available on select platforms and contains support for several features related to security and monitoring.

Packages

MongoDB provides packages of the officially supported MongoDB Enterprise builds in it's own repository. This repository provides the MongoDB Enterprise distribution in the following packages:

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

- `mongodb-enterprise`

  This package is a `metapackage` that will automatically install the four component packages listed below.

- `mongodb-enterprise-server`

  This package contains the `mongod` daemon and associated configuration and init scripts.

- `mongodb-enterprise-mongos`

  This package contains the `mongos` daemon.

- `mongodb-enterprise-shell`

  This package contains the `mongo` shell.

- `mongodb-enterprise-tools`

  This package contains the following MongoDB tools: `mongoimport bsondump`, `mongodump`, `mongoexport`, `mongofiles`, `mongoimport`, `mongooplog`, `mongoperf`, `mongorestore`, `mongostat`, and `mongotop`.

Control Scripts

The `mongodb-enterprise` package includes various *control scripts*, including the init script `/etc/rc.d/init.d/mongod`.

The package configures MongoDB using the `/etc/mongod.conf` file in conjunction with the control scripts. See the `Configuration File` reference for documentation of settings available in the configuration file.

As of version 2.6.11, there are no control scripts for `mongos`. The `mongos` process is used only in *sharding* (page 681). You can use the `mongod` init script to derive your own `mongos` control script.

Prerequisites

MongoDB only provides Enterprise packages for 64-bit builds of SUSE Enterprise Linux version 11.

Use the provided distribution packages as described in this page if possible. These packages will automatically install all of MongoDB's dependencies, and are the recommended installation method.

Install MongoDB Enterprise

Step 1: Configure the package management system (`zypper`). Add the repository so that you can install MongoDB using `zypper`.

Use the following command to specify the MongoDB 2.6 branch:

```
sudo zypper addrepo --no-gpgcheck https://repo.mongodb.com/zypper/suse/11/mongodb-enterprise/2.6/x86_
```

Step 2: Install the MongoDB packages and associated tools. To install the latest release of MongoDB 2.6, issue the following command:

```
sudo zypper install mongodb-enterprise
```

To install a specific release of MongoDB, specify each component package individually and append the version number to the package name, as in the following example:

```
sudo zypper install mongodb-enterprise-2.6.10 mongodb-enterprise-server-2.6.10 mongodb-enterprise-she
```

You can specify any available version of MongoDB. However `zypper` will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, pin the packages by running the following command:

```
sudo zypper addlock mongodb-enterprise-2.6.10 mongodb-enterprise-server-2.6.10 mongodb-enterprise-she
```

Install MongoDB Enterprise From Tarball

Note: The Enterprise tarball includes an example SNMP configuration file named `mongod.conf`. This file is not a MongoDB configuration file.

Step 1: Install dependencies

```
sudo zypper install cyrus-sasl krb5 libgcc46 libopenssl0_9_8 libsnmp15 libstdc++46 zlib
```

Step 2: Download and install the MongoDB Enterprise packages. After you have installed the required prerequisite packages, download and install the MongoDB Enterprise packages from http://mongodb.com/download/. The MongoDB binaries are located in the `bin/` directory of the archive. To download and install, use the following sequence of commands.

```
curl -O http://downloads.10gen.com/linux/mongodb-linux-x86_64-enterprise-suse11-2.6.11.tgz
tar -zxvf mongodb-linux-x86_64-enterprise-suse11-2.6.11.tgz
cp -R -n mongodb-linux-x86_64-enterprise-suse11-2.6.11/ mongodb
```

Step 3: Install the MongoDB packages and associated tools. Once you have copied the MongoDB binaries to their target location, ensure that the location is included in your `PATH` variable. If it is not, either include it or create symbolic links from the binaries to a directory that is included.

Run MongoDB Enterprise

The MongoDB instance stores its data files in `/var/lib/mongo` and its log files in `/var/log/mongodb` by default, and runs using the `mongod` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Start MongoDB. You can start the `mongod` process by issuing the following command:

```
sudo service mongod start
```

Step 2: Verify that MongoDB has started successfully You can verify that the `mongod` process has started successfully by checking the contents of the log file at `/var/log/mongodb/mongod.log` for a line reading

```
[initandlisten] waiting for connections on port <port>
```

where `<port>` is the port configured in `/etc/mongod.conf`, `27017` by default.

You can optionally ensure that MongoDB will start following a system reboot by issuing the following command:

```
sudo chkconfig mongod on
```

Step 3: Stop MongoDB. As needed, you can stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 4: Restart MongoDB. You can restart the `mongod` process by issuing the following command:

```
sudo service mongod restart
```

You can follow the state of the process for errors or important messages by watching the output in the `/var/log/mongodb/mongod.log` file.

Step 5: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Uninstall MongoDB

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs. The following section guides you through the necessary steps.

> **Warning:** This process will *completely* remove MongoDB, its configuration, and *all* databases. This process is not reversible, so ensure that all of your configuration and data is backed up before proceeding.

Step 1: Stop MongoDB. Stop the `mongod` process by issuing the following command:

```
sudo service mongod stop
```

Step 2: Remove Packages. Remove any MongoDB packages that you had previously installed.

```
sudo zypper remove $(rpm -qa | grep mongodb-enterprise)
```

Step 3: Remove Data Directories. Remove MongoDB databases and log files.

```
sudo rm -r /var/log/mongodb
sudo rm -r /var/lib/mongo
```

Install MongoDB Enterprise on Amazon Linux AMI

On this page

Overview

Use this tutorial to install MongoDB Enterprise[25] on Amazon Linux AMI. MongoDB Enterprise is available on select platforms and contains support for several features related to security and monitoring.

Prerequisites

To install all of MongoDB's dependencies, run the following command:

```
yum install cyrus-sasl cyrus-sasl-plain cyrus-sasl-gssapi krb5-libs \
            lm_sensors-libs net-snmp-agent-libs net-snmp openssl rpm-libs \
            tcp_wrappers-libs
```

Install MongoDB Enterprise

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

Step 1: Download and install the MongoDB Enterprise packages. After you have installed the required prerequisite packages, download and install the MongoDB Enterprise packages from http://mongodb.com/download/. The MongoDB binaries are located in the `bin/` directory of the archive. To download and install, use the following sequence of commands.

```
curl -O http://downloads.10gen.com/linux/mongodb-linux-x86_64-enterprise-amzn64-2.6.11.tgz
tar -zxvf mongodb-linux-x86_64-enterprise-amzn64-2.6.11.tgz
cp -R -n mongodb-linux-x86_64-enterprise-amzn64-2.6.11/ mongodb
```

Step 2: Ensure the location of the MongoDB binaries is included in the `PATH` variable. Once you have copied the MongoDB binaries to their target location, ensure that the location is included in your `PATH` variable. If it is not, either include it or create symbolic links from the binaries to a directory that is included.

Run MongoDB Enterprise

The MongoDB instance stores its data files in `/var/lib/mongo` and its log files in `/var/log/mongodb` by default, and runs using the `mongod` user account. You can specify alternate log and data file directories in `/etc/mongod.conf`. See `systemLog.path` and `storage.dbPath` for additional information.

If you change the user that runs the MongoDB process, you **must** modify the access control rights to the `/var/lib/mongo` and `/var/log/mongodb` directories to give this user access to these directories.

Step 1: Create the data directory. Before you start MongoDB for the first time, create the directory to which the `mongod` process will write data. By default, the `mongod` process uses the `/data/db` directory. If you create a directory other than this one, you must specify that directory in the `dbpath` option when starting the `mongod` process later in this procedure.

The following example command creates the default `/data/db` directory:

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

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

Step 2: Set permissions for the data directory. Before running `mongod` for the first time, ensure that the user account running `mongod` has read and write permissions for the directory.

Step 3: Run MongoDB. To run MongoDB, run the `mongod` process at the system prompt. If necessary, specify the path of the `mongod` or the data directory. See the following examples.

Run without specifying paths If your system `PATH` variable includes the location of the `mongod` binary and if you use the default data directory (i.e., `/data/db`), simply enter `mongod` at the system prompt:

```
mongod
```

Specify the path of the `mongod` If your `PATH` does not include the location of the `mongod` binary, enter the full path to the `mongod` binary at the system prompt:

```
<path to binary>/mongod
```

Specify the path of the data directory If you do not use the default data directory (i.e., `/data/db`), specify the path to the data directory using the *`--dbpath`* option:

```
mongod --dbpath <path to data directory>
```

Step 4: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Install MongoDB Enterprise on Windows

On this page

New in version 2.6.

Overview

Use this tutorial to install MongoDB Enterprise[26] on Windows systems. MongoDB Enterprise is available on select platforms and contains support for several features related to security and monitoring.

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

Prerequisites

MongoDB Enterprise Server for Windows requires Windows Server 2008 R2 or later. The `.msi` installer includes all other software dependencies and will automatically upgrade any older version of MongoDB installed using an `.msi` file.

Get MongoDB Enterprise

Step 1: Download MongoDB Enterprise for Windows. Download the latest production release of MongoDB Enterprise[27].

Install MongoDB Enterprise

Interactive Installation

Step 1: Install MongoDB Enterprise for Windows. In Windows Explorer, locate the downloaded MongoDB `.msi` file, which typically is located in the default `Downloads` folder. Double-click the `.msi` file. A set of screens will appear to guide you through the installation process.

You may specify an installation directory if you choose the "Custom" installation option.

Note: These instructions assume that you have installed MongoDB to `C:\mongodb`.

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 folder (e.g. `D:\test\mongodb`).

Unattended Installation You may install MongoDB unattended on Windows from the command line using `msiexec.exe`.

Step 1: Install MongoDB Enterprise for Windows. Open a shell in the directory containing the `.msi` installation binary of your choice and invoke:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="<installation directory>"
```

By default, this method installs the following MongoDB binaries: `mongod.exe`, `mongo.exe`, `mongodump.exe`, `mongorestore.exe`, `mongoimport.exe`, `mongoexport.exe`, `mongostat.exe`, and `mongotop.exe`. You can specify the installation location for the executable by modifying the `<installation directory>` value. To install specific subsets of the binaries, you may specify an `ADDLOCAL` argument:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="<installation directory>" ADDLOCAL=<k
```

The `<binary set(s)>` value is a comma-separated list including one or more of the following:

- `Server` - includes `mongod.exe`
- `Client` - includes `mongo.exe`
- `MonitoringTools` - includes `mongostat.exe` and `mongotop.exe`
- `ImportExportTools` - includes `mongodump.exe`, `mongorestore.exe`, `mongoexport.exe`, and `mongoimport.exe`)

[27] http://www.mongodb.com/products/mongodb-enterprise

- `MiscellaneousTools` - includes `bsondump.exe`, `mongofiles.exe`, `mongooplog.exe`, and `mongoperf.exe`

For instance, to install *only* the entire set of tools to `C:\mongodb`, invoke:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="C:\mongodb" ADDLOCAL=MonitoringTools,
```

You may also specify `ADDLOCAL=ALL` to install the complete set of binaries, as in the following:

```
msiexec.exe /q /i mongodb-<version>-signed.msi INSTALLLOCATION="C:\mongodb" ADDLOCAL=ALL
```

Run MongoDB Enterprise

Warning: Do not make `mongod.exe` visible on public networks without running in "Secure Mode" with the `auth` setting. MongoDB is designed to be run in trusted environments, and the database does not enable "Secure Mode" by default.

Step 1: Set up the MongoDB environment. MongoDB requires a *data directory* to store all data. MongoDB's default data directory path is `\data\db`. Create this folder using the following commands from a *Command Prompt*:

```
md \data\db
```

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

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

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

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

You may also specify the `dbpath` in a `configuration file`.

Step 2: Start MongoDB. To start MongoDB, run `mongod.exe`. For example, from the *Command Prompt*:

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

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

Depending on the security level of your system, Windows may pop up 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 see the *Security Documentation* (page 316).

Step 3: Connect to MongoDB. To connect to MongoDB through the `mongo.exe` shell, open another *Command Prompt*.

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

If you want to develop applications using .NET, see the documentation of C# and MongoDB[28] for more information.

[28] https://docs.mongodb.org/ecosystem/drivers/csharp

Step 4: Begin using MongoDB. To begin using MongoDB, see *Getting Started with MongoDB* (page 52). Also consider the *Production Notes* (page 210) document before deploying MongoDB in a production environment.

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

Configure a Windows Service for MongoDB Enterprise

Step 1: Open an Administrator command prompt.

Windows 7 / Vista / Server 2008 (and R2) Press `Win + R`, then type `cmd`, then press `Ctrl + Shift + Enter`.

Windows 8 Press `Win + X`, then press `A`.

Execute the remaining steps from the Administrator command prompt.

Step 2: Create directories. Create directories for your database and log files:

```
mkdir c:\data\db
mkdir c:\data\log
```

Step 3: Create a configuration file. Create a configuration file. The file **must** set `systemLog.path`. Include additional `configuration options` as appropriate.

For example, create a file at `C:\mongodb\mongod.cfg` that specifies both `systemLog.path` and `storage.dbPath`:

```
systemLog:
    destination: file
    path: c:\data\log\mongod.log
storage:
    dbPath: c:\data\db
```

Step 4: Install the MongoDB service.

Important: Run all of the following commands in *Command Prompt* with "Administrative Privileges".

Install the MongoDB service by starting `mongod.exe` with the `--install` option and the `-config` option to specify the previously created configuration file.

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

To use an alternate `dbpath`, specify the path in the configuration file (e.g. `C:\mongodb\mongod.cfg`) or on the command line with the *`--dbpath`* option.

If needed, you can install services for multiple instances of `mongod.exe` or `mongos.exe`. Install each service with a unique *`--serviceName`* and *`--serviceDisplayName`*. Use multiple instances only when sufficient system resources exist and your system design requires it.

Step 5: Start the MongoDB service.

```
net start MongoDB
```

Step 6: Stop or remove the MongoDB service as needed. To stop the MongoDB service use the following command:

```
net stop MongoDB
```

To remove the MongoDB service use the following command:

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

Manually Create a Windows Service for MongoDB Enterprise

You can set up the MongoDB server as a *Windows Service* that starts automatically at boot time.

The following procedure assumes you have installed MongoDB using the `.msi` installer with the path `C:\mongodb\`.

If you have installed in an alternative directory, you will need to adjust the paths as appropriate.

Step 1: Open an Administrator command prompt.

Windows 7 / Vista / Server 2008 (and R2) Press `Win + R`, then type `cmd`, then press `Ctrl + Shift + Enter`.

Windows 8 Press `Win + X`, then press `A`.

Execute the remaining steps from the Administrator command prompt.

Step 2: Create directories. Create directories for your database and log files:

```
mkdir c:\data\db
mkdir c:\data\log
```

Step 3: Create a configuration file. Create a configuration file. The file **must** set `systemLog.path`. Include additional `configuration options` as appropriate.

For example, create a file at `C:\mongodb\mongod.cfg` that specifies both `systemLog.path` and `storage.dbPath`:

```
systemLog:
    destination: file
    path: c:\data\log\mongod.log
storage:
    dbPath: c:\data\db
```

Step 4: Create the MongoDB service. Create the MongoDB service.

```
sc.exe create MongoDB binPath= "\"C:\mongodb\mongod.exe\" --service --config= \"C:\mongodb\mongod.cfg
```

`sc.exe` requires a space between "=" and the configuration values (eg "binPath= "), and a "\" to escape double quotes.

If successfully created, the following log message will display:

```
[SC] CreateService SUCCESS
```

Step 5: Start the MongoDB service.

```
net start MongoDB
```

Step 6: Stop or remove the MongoDB service as needed. To stop the MongoDB service, use the following command:

```
net stop MongoDB
```

To remove the MongoDB service, first stop the service and then run the following command:

```
sc.exe delete MongoDB
```

2.1.5 Verify Integrity of MongoDB Packages

On this page

Overview

The MongoDB release team digitally signs all software packages to certify that a particular MongoDB package is a valid and unaltered MongoDB release.

Before installing MongoDB, you can validate packages using either a PGP signature or with MD5 and SHA checksums of the MongoDB packages. The PGP signatures store an encrypted hash of the software package, that you can validate to ensure that the package you have is consistent with the official package release. MongoDB also publishes MD5 and SHA hashes of the official packages that you can use to confirm that you have a valid package.

Considerations

MongoDB signs each release branch with a different PGP key.

The public `.asc` and `.pub` key files for each branch are available for download. For example, the 2.2 keys are available at the following URLs:

```
https://www.mongodb.org/static/pgp/server-2.2.asc
https://www.mongodb.org/static/pgp/server-2.2.pub
```

Replace `2.2` with the appropriate release number to download public key. Keys are available for all MongoDB releases beginning with 2.2.

Procedures

Use PGP/GPG

Step 1: Download the MongoDB installation file. Download the binaries from `https://www.mongodb.org/downloads` based on your environment.

For example, to download the `2.6.0` release for OS X through the shell, type this command:

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.0.tgz
```

Step 2: Download the public signature file.

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.0.tgz.sig
```

Step 3: Download then import the key file. If you have not downloaded and imported the key file, enter these commands:

```
curl -LO https://www.mongodb.org/static/pgp/server-2.6.asc
gpg --import server-2.6.asc
```

You should receive this message:

```
gpg: key AAB2461C: public key "MongoDB 2.6 Release Signing Key <packaging@mongodb.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

Step 4: Verify the MongoDB installation file. Type this command:

```
gpg --verify mongodb-osx-x86_64-2.6.0.tgz.sig mongodb-osx-x86_64-2.6.0.tgz
```

You should receive this message:

```
gpg: Signature made Thu Mar  6 15:11:28 2014 EST using RSA key ID AAB2461C
gpg: Good signature from "MongoDB 2.6 Release Signing Key <packaging@mongodb.com>"
```

Download and import the key file, as described above, if you receive a message like this one:

```
gpg: Signature made Thu Mar  6 15:11:28 2014 EST using RSA key ID AAB2461C
gpg: Can't check signature: public key not found
```

`gpg` will return the following message if the package is properly signed, but you do not currently trust the signing key in your local `trustdb`.

```
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: DFFA 3DCF 326E 302C 4787  673A 01C4 E7FA AAB2 461C
```

Use SHA

MongoDB provides checksums using both the SHA-1 and SHA-256 hash functions. You can use either, as you like.

Step 1: Download the MongoDB installation file. Download the binaries from `https://www.mongodb.org/downloads` based on your environment.

For example, to download the `2.6.0` release for OS X through the shell, type this command:

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.0.tgz
```

Step 2: Download the SHA1 and SHA256 file.

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.3.tgz.sha1
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.3.tgz.sha256
```

Step 3: Use the SHA-256 checksum to verify the MongoDB package file. Compute the checksum of the package file:

```
shasum mongodb-linux-x86_64-2.6.3.tgz
```

which will generate this result:

```
fe511ee40428edda3a507f70d2b91d16b0483674 mongodb-osx-x86_64-2.6.3.tgz
```

Enter this command:

```
cat mongodb-linux-x86_64-2.6.3.tgz.sha1
```

which will generate this result:

```
fe511ee40428edda3a507f70d2b91d16b0483674 mongodb-osx-x86_64-2.6.3.tgz
```

The output of the `shasum` and `cat` commands should be identical.

Step 3: Use the SHA-1 checksum to verify the MongoDB package file. Compute the checksum of the package file:

```
shasum -a 256 mongodb-linux-x86_64-2.6.3.tgz
```

which will generate this result:

```
be3a5e9f4e9c8e954e9af7053776732387d2841a019185eaf2e52086d4d207a3  mongodb-osx-x86_64-2.6.3.tgz
```

Enter this command:

```
cat mongodb-linux-x86_64-2.6.3.tgz.sha256
```

which will generate this result:

```
be3a5e9f4e9c8e954e9af7053776732387d2841a019185eaf2e52086d4d207a3  mongodb-osx-x86_64-2.6.3.tgz
```

The output of the `shasum` and `cat` commands should be identical.

Use MD5

Step 1: Download the MongoDB installation file. Download the binaries from `https://www.mongodb.org/downloads` based on your environment.

For example, to download the `2.6.0` release for OS X through the shell, type this command:

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.0.tgz
```

Step 2: Download the MD5 file.

```
curl -LO http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.6.0.tgz.md5
```

Step 3: Verify the checksum values for the MongoDB package file (Linux). Compute the checksum of the package file:

```
md5 mongodb-linux-x86_64-2.6.0.tgz
```

which will generate this result:

```
MD5 (mongodb-linux-x86_64-2.6.0.tgz) = a937d49881f90e1a024b58d642011dc4
```

Enter this command:

```
cat mongodb-linux-x86_64-2.6.0.tgz.md5
```

which will generate this result:

```
a937d49881f90e1a024b58d642011dc4
```

The output of the `md5` and `cat` commands should be identical.

Step 4: Verify the MongoDB installation file (OS X). Compute the checksum of the package file:

```
md5sum -c mongodb-osx-x86_64-2.6.0.tgz.md5 mongodb-osx-x86_64-2.6.0.tgz
```

which will generate this result:

```
mongodb-osx-x86_64-2.6.0-rc1.tgz ok
```

2.2 First Steps with MongoDB

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

Getting Started with MongoDB **(page 52)** An introduction to the basic operation and use of MongoDB.

Generate Test Data **(page 57)** To support initial exploration, generate test data to facilitate testing.

2.2.1 Getting Started with MongoDB

On this page

This tutorial provides an introduction to basic database operations using the `mongo` shell. `mongo` is a part of the standard MongoDB distribution and provides a full JavaScript environment with complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB. See the mongo JavaScript API[29] documentation and the `mongo` shell `JavaScript Method Reference`.

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.

[29] https://api.mongodb.org/js

For instructions on installing MongoDB and starting the database server, see the appropriate *installation* (page 5) document.

Connect to a Database

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

Connect to a `mongod`

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

```
mongo
```

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

Select a Database

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

```
db
```

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

   ```
   show dbs
   ```

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

   ```
   use mydb
   ```

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

   ```
   db
   ```

   At this point, if you issue the `show dbs` operation again, it will not include the `mydb` database. MongoDB will not permanently create a database until you insert data into that database. The *Create a Collection and Insert Documents* (page 54) section describes the process for inserting data.

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

Display `mongo` Help

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

```
help
```

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

Create a Collection and Insert Documents

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

MongoDB will create a collection implicitly upon its first use. You do not need to create a collection before inserting data. Furthermore, because MongoDB uses *dynamic schemas* (page 762), you also need not specify the structure of your documents before inserting them into the collection.

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

   ```
   db
   ```

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

   ```
   use mydb
   ```

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

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

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

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

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

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

   ```
   show collections
   ```

   The `mongo` shell will return the list of the collections in the current (i.e. `mydb`) database. At this point, the only collection with user data is `testData`.

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

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

   This operation returns the following results. The *ObjectId* (page 184) values will be unique:

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

   All MongoDB documents must have an `_id` field with a unique value. These operations do not explicitly specify a value for the `_id` field, so `mongo` creates a unique *ObjectId* (page 184) value for the field before inserting it into the collection.

Insert Documents using a For Loop or a JavaScript Function

To perform the remaining procedures in this tutorial, first add more documents to your database using one or both of the procedures described in *Generate Test Data* (page 57).

Working with the Cursor

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

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

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

Iterate over the Cursor with a Loop

Before using this procedure, add documents to a collection using one of the procedures in *Generate Test Data* (page 57). 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 all documents:

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

Use Array Operations with the Cursor

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

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

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

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

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

   MongoDB returns the following:

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

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

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

For more information on the cursor, see *crud-read-cursor*.

Query for Specific Documents

MongoDB has a rich query system that allows you to select and filter the documents in a collection along specific fields and values. See *Query Documents* (page 100) and *Read Operations* (page 64) for a full account of queries in MongoDB.

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

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

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

MongoDB returns one document that fits this criteria:

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

Return a Single Document from a Collection

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

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

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

For more information on querying for documents, see the *Query Documents* (page 100) and *Read Operations* (page 64) documentation.

Limit the Number of Documents in the Result Set

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

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

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

MongoDB will return the following result, with different *ObjectId* (page 184) values:

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

Next Steps with MongoDB

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

- *MongoDB CRUD Operations* (page 61)
- *SQL to MongoDB Mapping Chart* (page 136)
- MongoDB Drivers[30]

[30] https://docs.mongodb.org/ecosystem/drivers

Additional Resources

- MongoDB University: Free, Online Courses for Developers and DBAs[31]
- MongoDB Architecture Guide[32]
- MongoDB Administration 101 Presentation[33]

2.2.2 Generate Test Data

On this page

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

Insert Multiple Documents Using a For Loop

Step 1: Insert new documents into the `testData` collection.

From the `mongo` shell, use the `for` loop. If the `testData` collection does not exist, MongoDB will implicitly create the collection.

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

Step 2: Query the collection.

Use `find()` to query the collection:

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

The `mongo` shell displays the first 20 documents in the collection. Your *ObjectId* (page 184) values will be different:

```
{ "_id" : ObjectId("53d7be30242b692a1138ac7d"), "x" : 1 }
{ "_id" : ObjectId("53d7be30242b692a1138ac7e"), "x" : 2 }
{ "_id" : ObjectId("53d7be30242b692a1138ac7f"), "x" : 3 }
{ "_id" : ObjectId("53d7be30242b692a1138ac80"), "x" : 4 }
{ "_id" : ObjectId("53d7be30242b692a1138ac81"), "x" : 5 }
{ "_id" : ObjectId("53d7be30242b692a1138ac82"), "x" : 6 }
{ "_id" : ObjectId("53d7be30242b692a1138ac83"), "x" : 7 }
{ "_id" : ObjectId("53d7be30242b692a1138ac84"), "x" : 8 }
{ "_id" : ObjectId("53d7be30242b692a1138ac85"), "x" : 9 }
{ "_id" : ObjectId("53d7be30242b692a1138ac86"), "x" : 10 }
{ "_id" : ObjectId("53d7be30242b692a1138ac87"), "x" : 11 }
{ "_id" : ObjectId("53d7be30242b692a1138ac88"), "x" : 12 }
{ "_id" : ObjectId("53d7be30242b692a1138ac89"), "x" : 13 }
```

[31] https://education.mongodb.com/?jmp=docs
[32] https://www.mongodb.com/lp/whitepaper/architecture-guide?jmp=docs
[33] http://www.mongodb.com/presentations/webinar-mongodb-administration-101?jmp=docs

```
{ "_id" : ObjectId("53d7be30242b692a1138ac8a"), "x" : 14 }
{ "_id" : ObjectId("53d7be30242b692a1138ac8b"), "x" : 15 }
{ "_id" : ObjectId("53d7be30242b692a1138ac8c"), "x" : 16 }
{ "_id" : ObjectId("53d7be30242b692a1138ac8d"), "x" : 17 }
{ "_id" : ObjectId("53d7be30242b692a1138ac8e"), "x" : 18 }
{ "_id" : ObjectId("53d7be30242b692a1138ac8f"), "x" : 19 }
{ "_id" : ObjectId("53d7be30242b692a1138ac90"), "x" : 20 }
Type "it" for more
```

Step 3: Iterate through the cursor.

The `find()` method returns a cursor. To *iterate the cursor* (page 115) and return more documents, type `it` in the `mongo` shell. The shell will exhaust the cursor and return these documents:

```
{ "_id" : ObjectId("53d7be30242b692a1138ac91"), "x" : 21 }
{ "_id" : ObjectId("53d7be30242b692a1138ac92"), "x" : 22 }
{ "_id" : ObjectId("53d7be30242b692a1138ac93"), "x" : 23 }
{ "_id" : ObjectId("53d7be30242b692a1138ac94"), "x" : 24 }
{ "_id" : ObjectId("53d7be30242b692a1138ac95"), "x" : 25 }
```

Insert Multiple Documents with a `mongo` Shell Function

You can create a JavaScript function in your shell session to generate the above data. The `insertData()` JavaScript function that follows 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 set to an incremented integer, as in the following example documents:

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

Store the function in your *.mongorc.js* file. The `mongo` shell loads and parses the *.mongorc.js* file on startup so your function is available 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.

Additional Resources

- Python utils to create random JSON data and import into mongoDB[34]

See also:

MongoDB CRUD Concepts (page 64) and *Data Models* (page 149).

2.3 Additional Resources

- Install MongoDB using MongoDB Cloud Manager[35]
- *MongoDB CRUD Concepts* (page 64)
- *Data Models* (page 149)

[34] https://github.com/10gen-labs/ipsum
[35] https://docs.cloud.mongodb.com/tutorial/getting-started?jmp=docs

CHAPTER 3

MongoDB CRUD Operations

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

MongoDB CRUD Introduction **(page 61)** An introduction to the MongoDB data model as well as queries and data manipulations.

MongoDB CRUD Concepts **(page 64)** The core documentation of query and data manipulation.

MongoDB CRUD Tutorials **(page 96)** Examples of basic query and data modification operations.

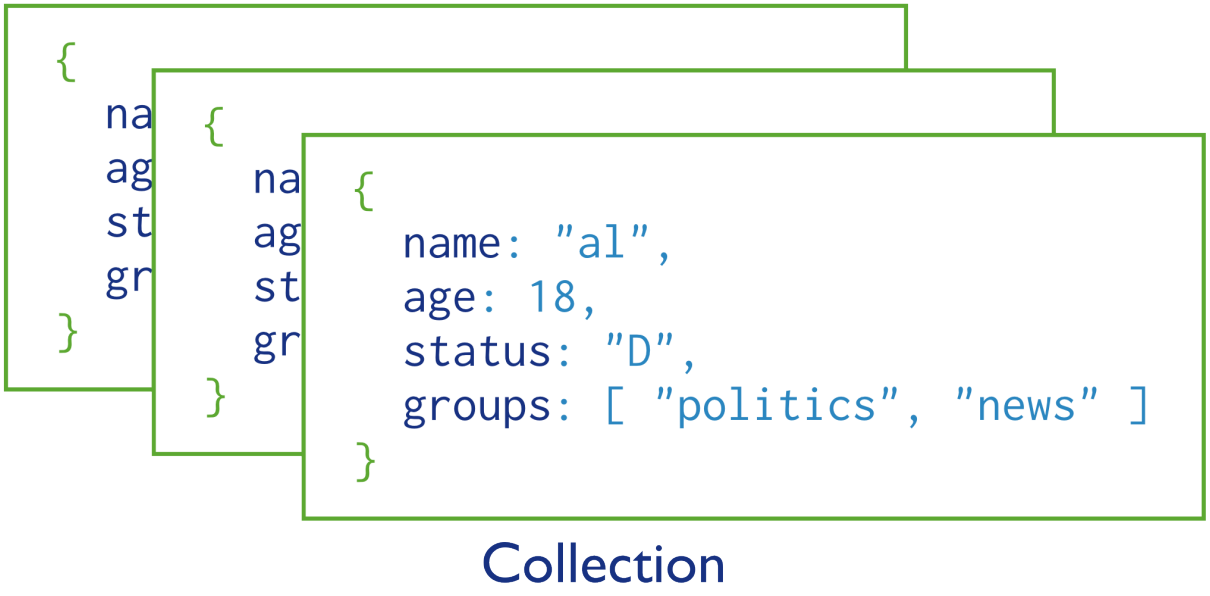
MongoDB CRUD Reference **(page 134)** Reference material for the query and data manipulation interfaces.

3.1 MongoDB CRUD Introduction

On this page

MongoDB stores data in the form of *documents*, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values (e.g. dictionaries, hashes, maps, and associative arrays). Formally, MongoDB documents are *BSON* documents. BSON is a binary representation of *JSON* with additional type information. In the documents, the value of a field can be any of the BSON data types, including other documents, arrays, and arrays of documents. For more information, see *Documents* (page 176).

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



3.1.1 Database Operations

Query

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

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

See *Read Operations Overview* (page 65) for more information.

Data Modification

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

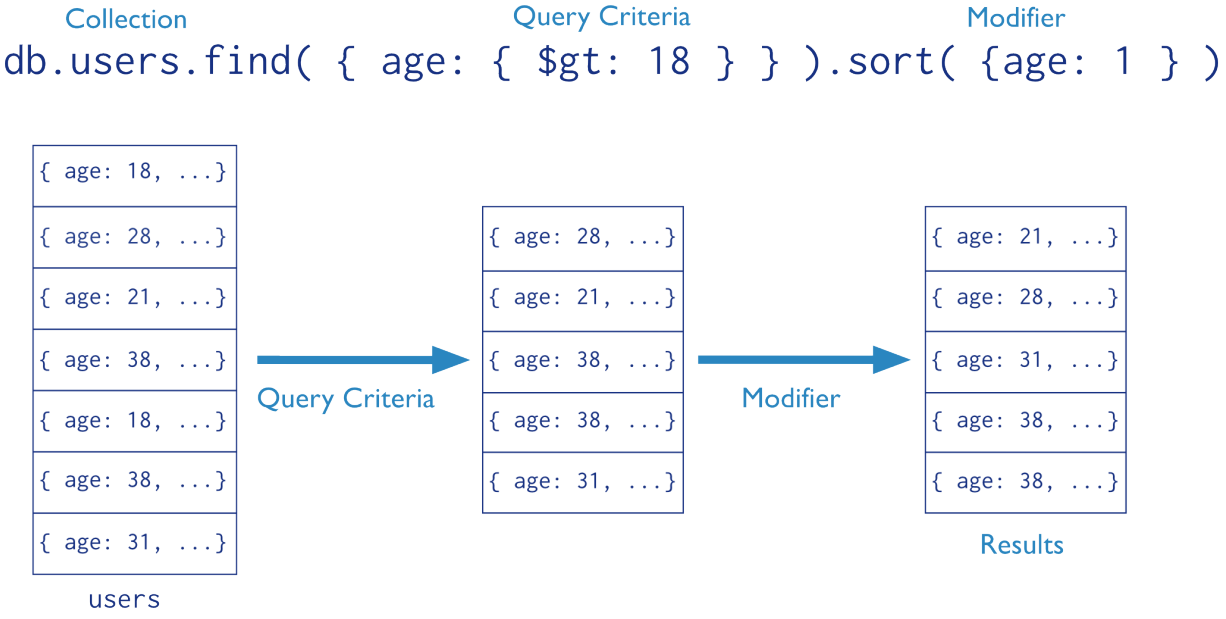
In the following diagram, the insert operation adds a new document to the `users` collection.

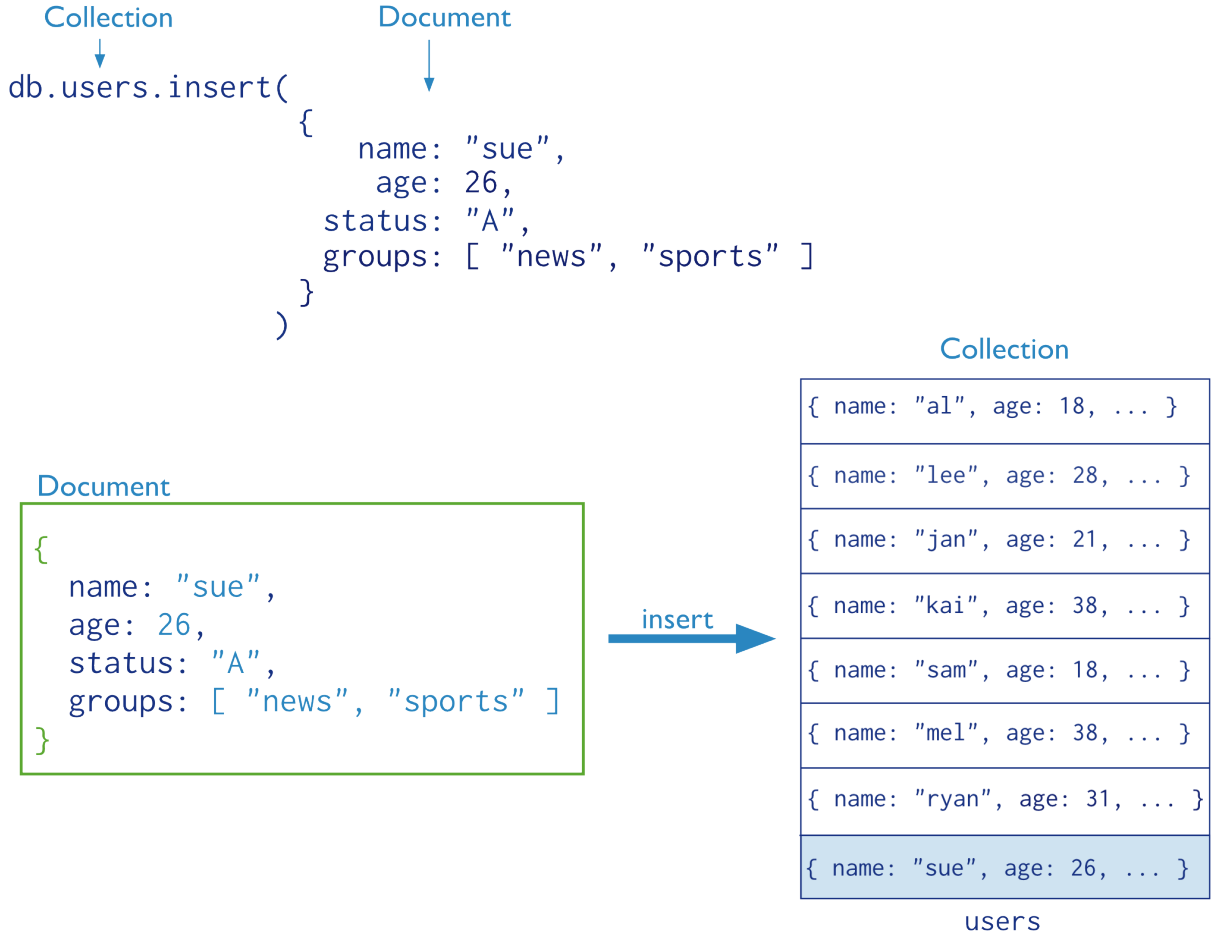
See *Write Operations Overview* (page 78) for more information.

3.1.2 Related Features

Indexes

To enhance the performance of common queries and updates, MongoDB has full support for secondary indexes. These indexes allow applications to store a *view* of a portion of the collection in an efficient data structure. Most indexes store


Collection
Query Criteria
Modifier
db.users.find({ age: { $gt: 18 } }).sort({age: 1 })
{ age: 18, ...}
{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 18, ...}
{ age: 38, ...}
{ age: 31, ...}
users
Query Criteria
{ age: 28, ...}
{ age: 21, ...}
{ age: 38, ...}
{ age: 38, ...}
{ age: 31, ...}
Modifier
{ age: 21, ...}
{ age: 28, ...}
{ age: 31, ...}
{ age: 38, ...}
{ age: 38, ...}
Results


Collection
Document
db.users.insert(
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
)
Collection
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
users
Document
{
name: "sue",
age: 26,
status: "A",
groups: ["news", "sports"]
}
insert

an ordered representation of all values of a field or a group of fields. Indexes may also *enforce uniqueness* (page 506), store objects in a *geospatial representation* (page 494), and facilitate *text search* (page 501).

Replica Set Read Preference

For replica sets and sharded clusters with replica set components, applications specify *read preferences* (page 591). A read preference determines how the client directs read operations to the set.

Write Concern

Applications can also control the behavior of write operations using *write concern* (page 82). Particularly useful for deployments with replica sets, the write concern semantics allow clients to specify the assurance that MongoDB provides when reporting on the success of a write operation.

Aggregation

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

3.2 MongoDB CRUD Concepts

The *Read Operations* (page 64) and *Write Operations* (page 77) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

Read Operations **(page 64)** Queries are the core operations that return data in MongoDB. Introduces queries, their behavior, and performances.

- ***Cursors*** **(page 68)** Queries return iterable objects, called cursors, that hold the full result set.
- ***Query Optimization*** **(page 70)** Analyze and improve query performance.
- ***Distributed Queries*** **(page 74)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Write Operations **(page 77)** Write operations insert, update, or remove documents in MongoDB. Introduces data create and modify operations, their behavior, and performances.

- ***Write Concern*** **(page 82)** Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.
- ***Distributed Write Operations*** **(page 87)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

Continue reading from *Write Operations* (page 77) for additional background on the behavior of data modification operations in MongoDB.

3.2.1 Read Operations

The following documents describe read operations:

Read Operations Overview **(page 65)** A high level overview of queries and projections in MongoDB, including a discussion of syntax and behavior.

Cursors **(page 68)** Queries return iterable objects, called cursors, that hold the full result set.

Query Optimization **(page 70)** Analyze and improve query performance.

Query Plans **(page 72)** MongoDB executes queries using optimal *plans*.

Distributed Queries **(page 74)** Describes how *sharded clusters* and *replica sets* affect the performance of read operations.

Read Operations Overview

On this page

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

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

Query Interface

For query operations, MongoDB provides a `db.collection.find()` method. The method accepts both the query criteria and projections and returns a *cursor* (page 68) to the matching documents. You can optionally modify the query to impose limits, skips, and sort orders.

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

The next diagram shows the same query in SQL:

Example

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

This query selects the documents in the `users` collection that match the condition `age` is greater than `18`. To specify the greater than condition, query criteria uses the greater than (i.e. `$gt`) *query selection operator*. The query returns at most `5` matching documents (or more precisely, a cursor to those documents). The matching documents will return with only the `_id`, `name` and `address` fields. See *Projections* (page 67) for details.

See

SQL to MongoDB Mapping Chart (page 136) for additional examples of MongoDB queries and the corresponding SQL statements.

Query Behavior

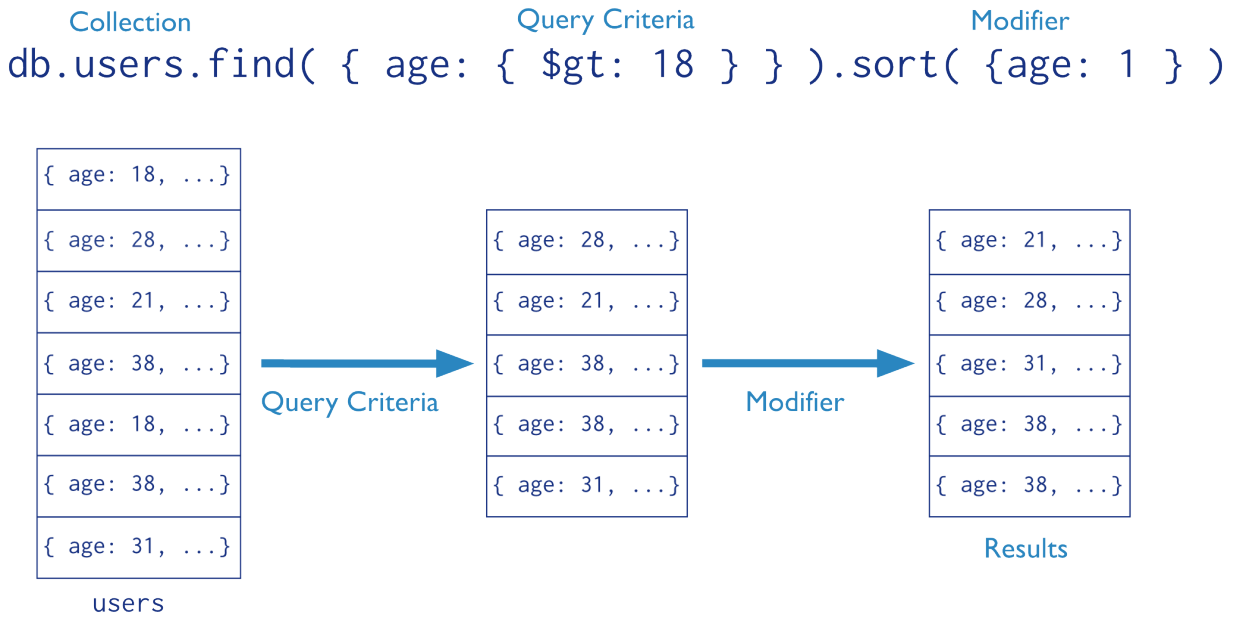
MongoDB queries exhibit the following behavior:

- All queries in MongoDB address a *single* collection.
- You can modify the query to impose `limits`, `skips`, and `sort orders`.
- The order of documents returned by a query is not defined unless you specify a `sort()`.
- Operations that *modify existing documents* (page 107) (i.e. *updates*) use the same query syntax as queries to select documents to update.
- In *aggregation* (page 439) pipeline, the `$match` pipeline stage provides access to MongoDB queries.

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

Query Statements

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



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

For additional examples of queries, see *Query Documents* (page 100).

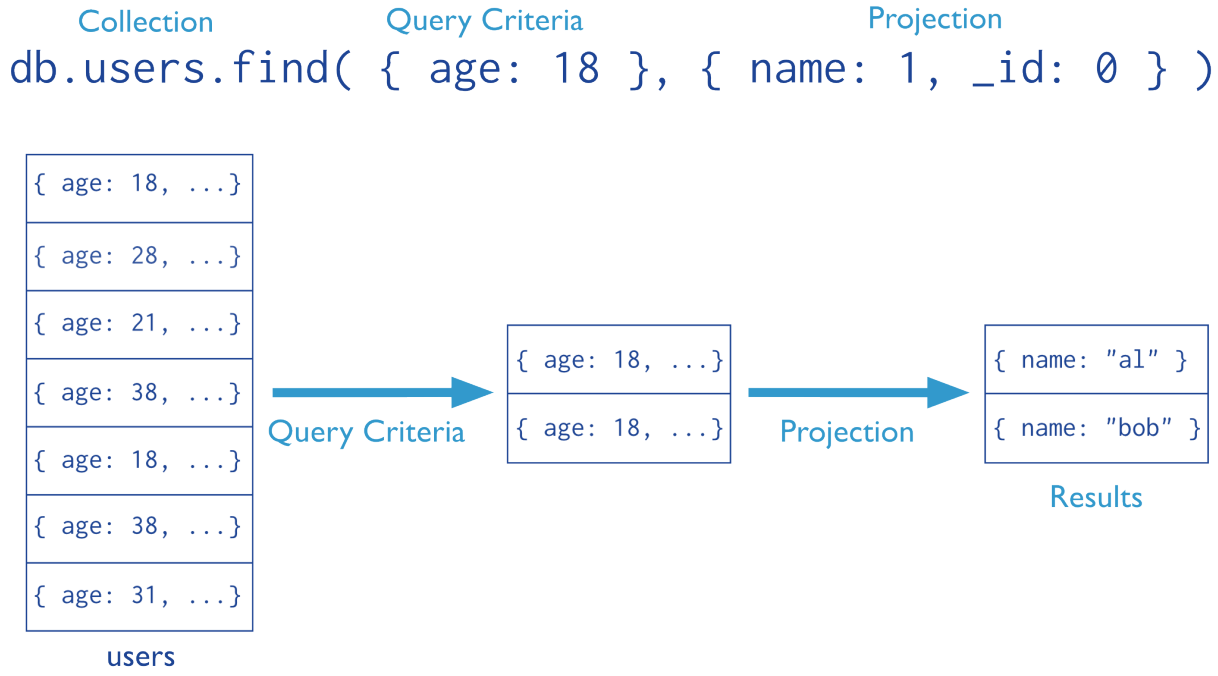
Projections

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

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

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

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



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

Projection Examples

Exclude One Field From a Result Set

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

This query selects documents in the `records` collection that match the condition `{ "user_id": { $lt: 42 } }`, and uses the projection `{ "history": 0 }` to exclude the `history` field from the documents in the result set.

Return Two fields *and* the `_id` Field

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

This query selects documents in the `records` collection that match the query `{ "user_id": { $lt: 42 } }` and uses the projection `{ "name": 1, "email": 1 }` to return just the `_id` field (implicitly included), `name` field, and the `email` field in the documents in the result set.

Return Two Fields *and* Exclude `_id`

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

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

See

Limit Fields to Return from a Query (page 112) for more examples of queries with projection statements.

Projection Behavior MongoDB projections have the following properties:

- By default, the `_id` field is included in the results. To suppress the `_id` field from the result set, specify `_id: 0` in the projection document.
- For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, and `$`.
- For related projection functionality in the *aggregation framework* (page 439) pipeline, use the `$project` pipeline stage.

Cursors

On this page

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

To access the documents, you need to iterate the cursor. However, in the `mongo` shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times [1] to print up to the first 20 documents in the results.

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

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

To manually iterate the cursor to access the documents, see *Iterate a Cursor in the mongo Shell* (page 115).

[1] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Executing Queries* (page 285) for more information.

Cursor Behaviors

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

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

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

Cursor Isolation Because the cursor is not isolated during its lifetime, intervening write operations on a document may result in a cursor that returns a document more than once if that document has changed. To handle this situation, see the information on *snapshot mode* (page 773).

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

For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort before returning any results.

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

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

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

myCursor.objsLeftInBatch();
```

Cursor Information

The `db.serverStatus()` method returns a document that includes a `metrics` field. The `metrics` field contains a `cursor` field with the following information:

- number of timed out cursors since the last server restart
- number of open cursors with the option `DBQuery.Option.noTimeout` set to prevent timeout after a period of inactivity
- number of "pinned" open cursors
- total number of open cursors

Consider the following example which calls the `db.serverStatus()` method and accesses the `metrics` field from the results and then the `cursor` field from the `metrics` field:

```
db.serverStatus().metrics.cursor
```

The result is the following document:

```
{
   "timedOut" : <number>
   "open" : {
      "noTimeout" : <number>,
      "pinned" : <number>,
      "total" : <number>
   }
}
```

See also:

`db.serverStatus()`

Query Optimization

On this page

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

Create an Index to Support Read Operations

If your application queries a collection on a particular field or set of fields, then an index on the queried field or a *compound index* (page 489) on the set of fields can prevent the query from scanning the whole collection to find and return the query results. For more information about indexes, see the *complete documentation of indexes in MongoDB* (page 485).

Example

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

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

To improve the performance of this query, add an ascending, or a descending, index to the `inventory` collection on the `type` field. [2] In the `mongo` shell, you can create indexes using the `db.collection.ensureIndex()` method:

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

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

To analyze the performance of the query with an index, see *Analyze Query Performance* (page 117).

In addition to optimizing read operations, indexes can support sort operations and allow for a more efficient storage utilization. See `db.collection.ensureIndex()` and *Indexing Tutorials* (page 519) for more information about index creation.

[2] For single-field indexes, the selection between ascending and descending order is immaterial. For compound indexes, the selection is important. See *indexing order* (page 490) for more details.

Query Selectivity

Query selectivity refers to how well the query predicate excludes or filters out documents in a collection. Query selectivity can determine whether or not queries can use indexes effectively or even use indexes at all.

More selective queries match a smaller percentage of documents. For instance, an equality match on the unique `_id` field is highly selective as it can match at most one document.

Less selective queries match a larger percentage of documents. Less selective queries cannot use indexes effectively or even at all.

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

The selectivity of `regular expressions` depends on the expressions themselves. For details, see *regular expression and index use*.

Covered Query

An index *covers* (page 71) a query when both of the following apply:

- all the fields in the *query* (page 100) are part of an index, **and**
- all the fields returned in the results are in the same index.

For example, a collection `inventory` has the following index on the `type` and `item` fields:

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

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

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

For the specified index to cover the query, the projection document must explicitly specify `_id: 0` to exclude the `_id` field from the result since the index does not include the `_id` field.

Performance Because the index contains all fields required by the query, MongoDB can both match the *query conditions* (page 100) and return the results using only the index.

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.

Limitations An index **cannot** cover a query if:

- the query is on a *sharded* collection and run against a *mongos*.

  Changed in version 2.6.4: In earlier versions, an index cannot cover a query on a sharded collection when run against either a `mongos` or the primary.

- 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 491) and cannot support a covered query.

- any of the indexed field in the query predicate or returned in the projection are fields in embedded documents. [3] For example, consider a collection `users` with documents of the following form:

  ```
  { _id: 1, user: { login: "tester" } }
  ```

  The collection has the following index:

  ```
  { "user.login": 1 }
  ```

  The `{ "user.login": 1 }` index does **not** cover the following query:

  ```
  db.users.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
  ```

  However, the query can use the `{ "user.login": 1 }` index to find matching documents.

`indexOnly` To determine whether a query is a covered query, use the `explain()` method. If the `explain()` output displays `true` for the `indexOnly` field, an index covers the query, and MongoDB queries only that index to match the query **and** return the results.

For more information see *Measure Index Use* (page 532).

Query Plans

On this page

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 only caches the plans for those query shapes that can have more than one viable plan.

The query optimizer occasionally reevaluates query plans as the content of the collection changes to ensure optimal query plans. You can also specify which indexes the optimizer evaluates with *Index Filters* (page 73).

You can use the `explain()` method to view statistics about the query plan for a given query. This information can help as you develop *indexing strategies* (page 551).

Query Optimization

To create a new query plan, the query optimizer:

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

[3] To index fields in embedded documents, use *dot notation*.

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

3. stops the testing of candidate plans and selects an index when one of the following events occur:
   - An *unordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned all the matching results; *or*
   - An *ordered query plan* has returned a threshold number of matching results:
     - Version 2.0: Threshold is the query batch size. The default batch size is 101.
     - Version 2.2: Threshold is 101.

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

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

Query Plan Revision

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

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

Changed in version 2.6: `explain()` operations no longer read from or write to the query planner cache.

Cached Query Plan Interface

New in version 2.6.

MongoDB provides `http://docs.mongodb.org/manual/reference/method/js-plan-cache` to view and modify the cached query plans.

Index Filters

New in version 2.6.

Index filters determine which indexes the optimizer evaluates for a *query shape*. A query shape consists of a combination of query, sort, and projection specifications. If an index filter exists for a given query shape, the optimizer only considers those indexes specified in the filter.

When an index filter exists for the query shape, MongoDB ignores the `hint()`. To see whether MongoDB applied an index filter for a query, check the `explain.filterSet` field of the `explain()` output.

Index filters only affects which indexes the optimizer evaluates; the optimizer may still select the collection scan as the winning plan for a given query shape.

Index filters exist for the duration of the server process and do not persist after shutdown. MongoDB also provides a command to manually remove filters.

Because index filters overrides the expected behavior of the optimizer as well as the `hint()` method, use index filters sparingly.

See `planCacheListFilters`, `planCacheClearFilters`, and `planCacheSetFilter`.

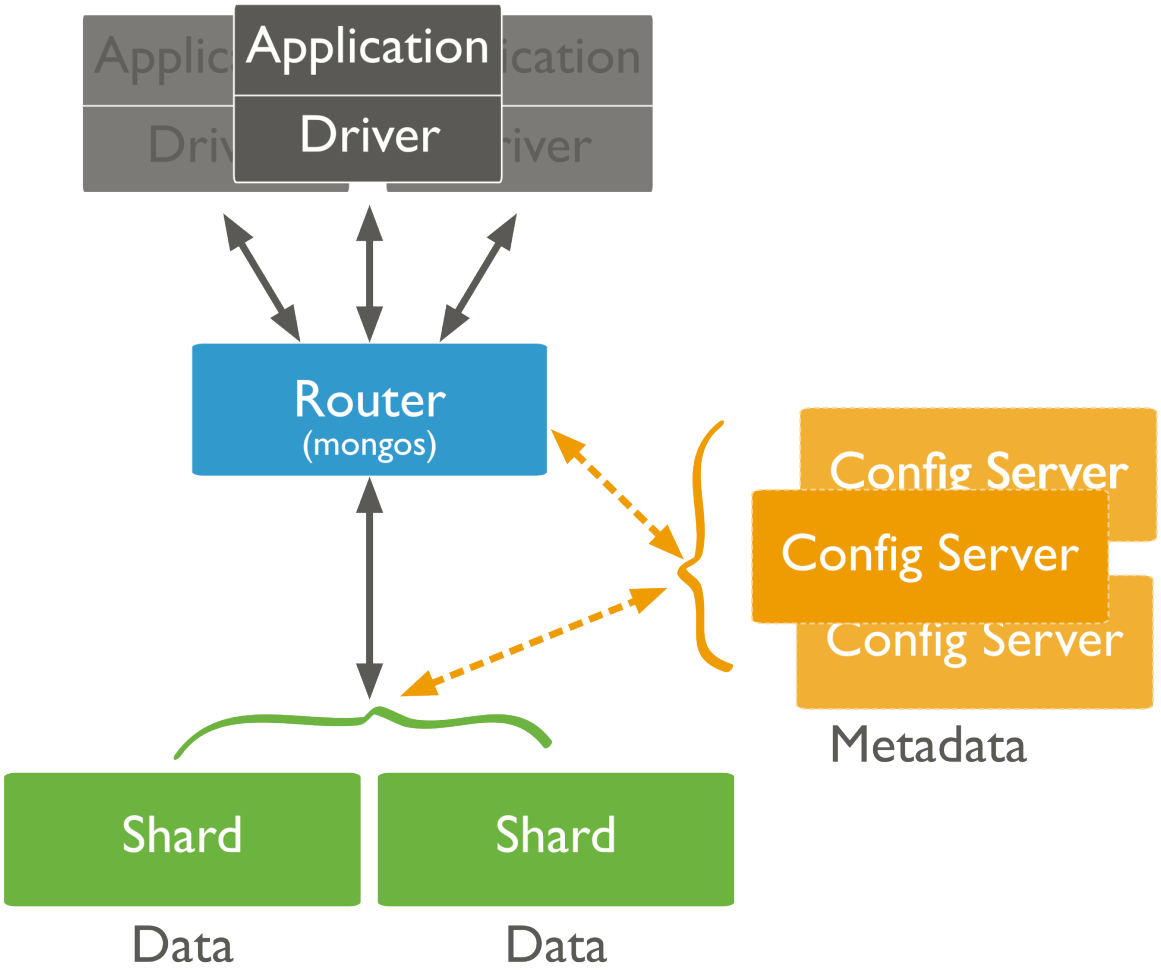
Distributed Queries

On this page

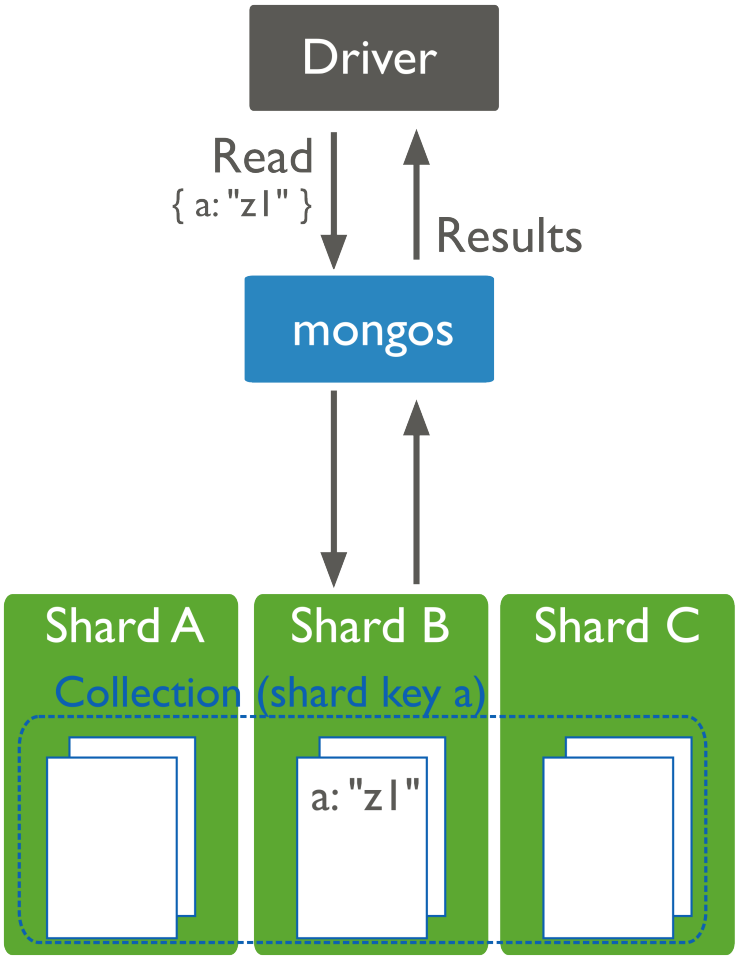
Read Operations to Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of `mongod` instances in a way that is nearly transparent to the application. For an overview of sharded clusters, see the *Sharding* (page 675) section of this manual.

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



Read operations on sharded clusters are most efficient when directed to a specific shard. Queries to sharded collections should include the collection's *shard key* (page 687). When a query includes a shard key, the `mongos` can use cluster metadata from the *config database* (page 684) to route the queries to shards.



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

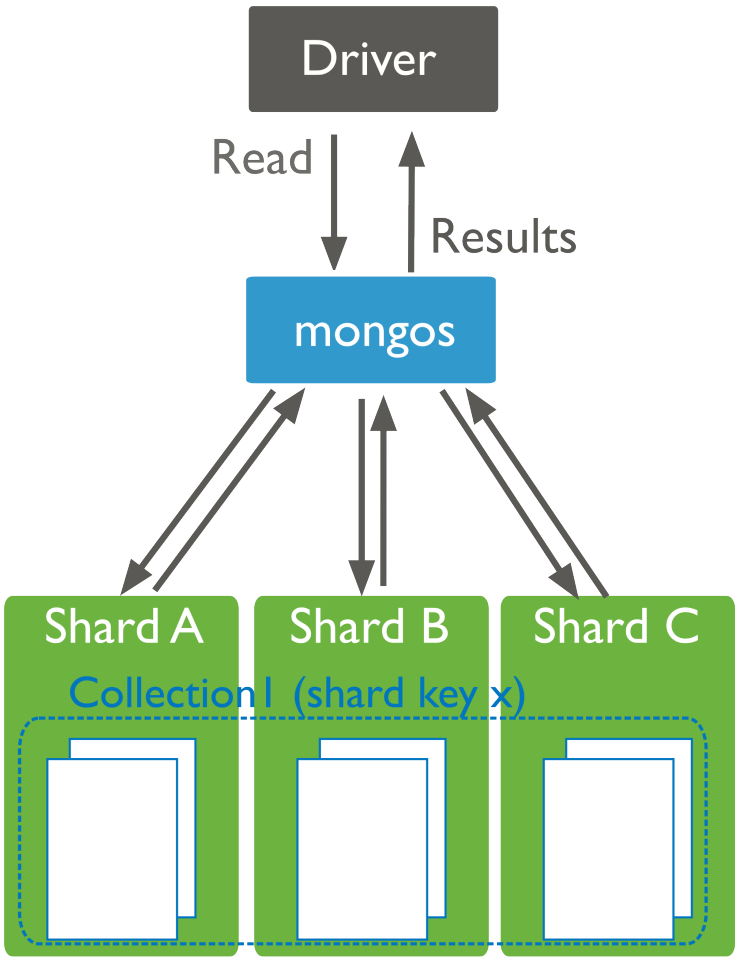
For more information on read operations in sharded clusters, see the *Sharded Cluster Query Routing* (page 692) and *Shard Keys* (page 687) sections.

Read Operations to Replica Sets

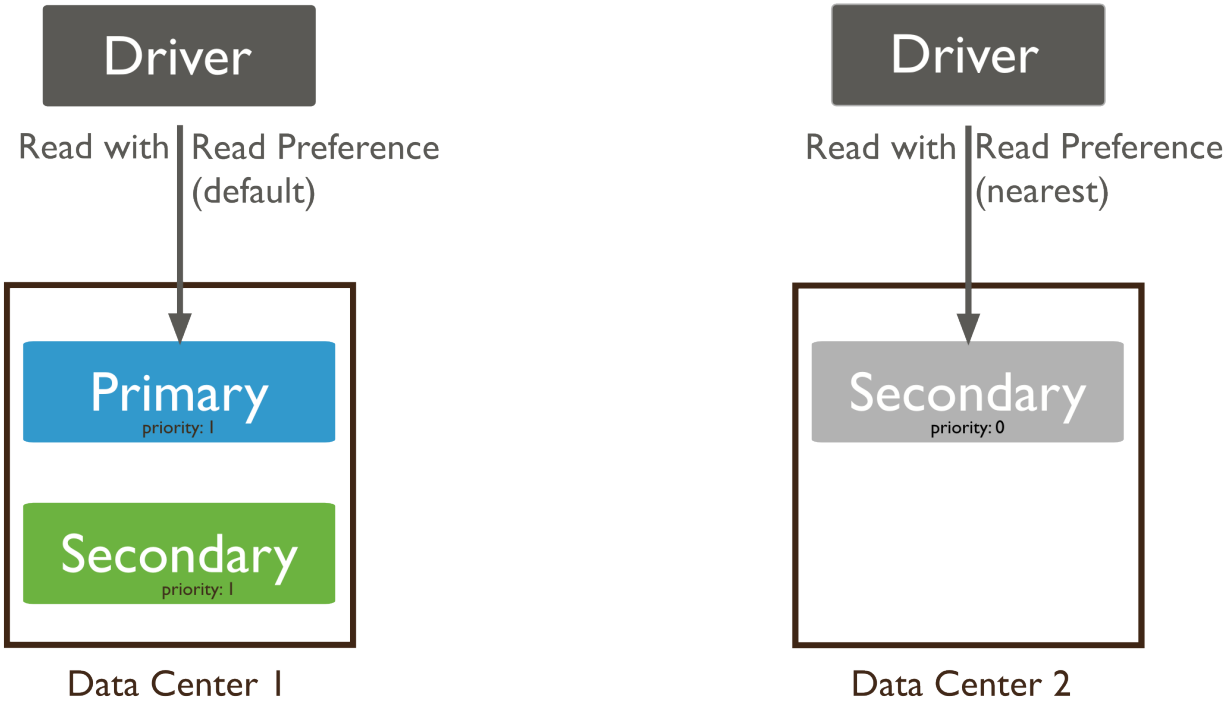
Replica sets use *read preferences* to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can modify that behavior by changing the *read preference mode* (page 670).

You can configure the *read preference mode* (page 670) 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),


Driver
Read
Results
mongos
Shard A
Shard B
Shard C
Collection 1 (shard key x)

- for backup operations, and/or
- to allow reads during *failover* (page 583) 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 trails the primary by some amount of time. [4]

For more information on read preference or on the read preference modes, see *Read Preference* (page 591) and *Read Preference Modes* (page 670).

3.2.2 Write Operations

The following documents describe write operations:

Write Operations Overview **(page 78)** Provides an overview of MongoDB's data insertion and modification operations, including aspects of the syntax, and behavior.

Write Concern **(page 82)** Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

Atomicity and Transactions **(page 86)** Describes write operation atomicity in MongoDB.

Distributed Write Operations **(page 87)** Describes how MongoDB directs write operations on *sharded clusters* and *replica sets* and the performance characteristics of these operations.

Write Operation Performance **(page 88)** Introduces the performance constraints and factors for writing data to MongoDB deployments.

Bulk Write Operations **(page 92)** Provides an overview of MongoDB's bulk write operations.

[4] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).

Storage **(page 94)** Introduces the storage allocation strategies available for MongoDB collections.

Write Operations Overview

On this page

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

There are three classes of write operations in MongoDB: *insert* (page 78), *update* (page 79), and *remove* (page 80). Insert operations add new data to a collection. Update operations modify existing data, and remove operations delete data from a collection. No insert, update, or remove can affect more than one document atomically.

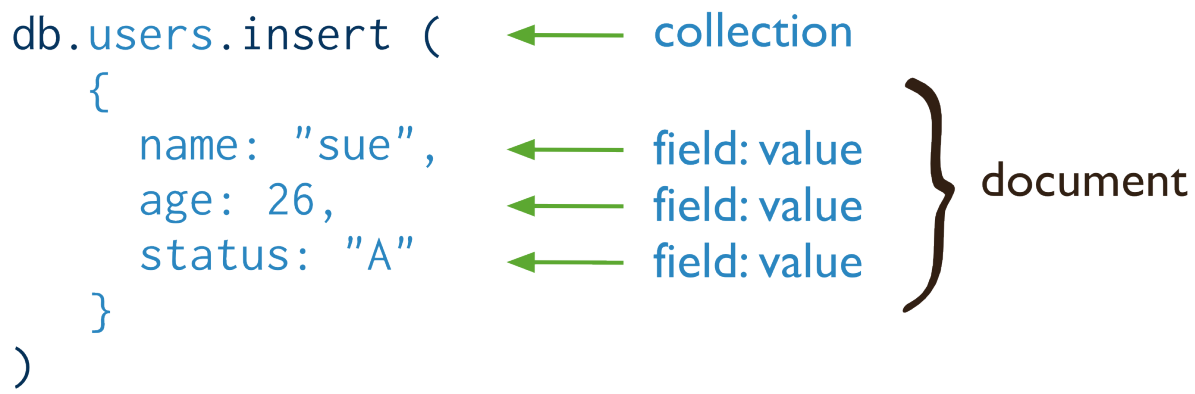
For the update and remove operations, you can specify criteria, or conditions, that identify the documents to update or remove. These operations use the same query syntax to specify the criteria as *read operations* (page 64).

MongoDB allows applications to determine the acceptable level of acknowledgement required of write operations. See *Write Concern* (page 82) for more information.

Insert

In MongoDB, the `db.collection.insert()` method adds new *documents* to a collection.

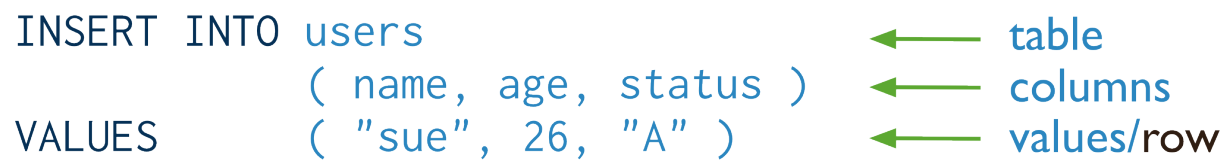
The following diagram highlights the components of a MongoDB insert operation:



The following diagram shows the same query in SQL:

Example

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




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

For more information and examples, see `db.collection.insert()`.

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

If you specify the `_id` field, the value must be unique within the collection. For operations with *write concern* (page 82), if you try to create a document with a duplicate `_id` value, `mongod` returns a duplicate key exception.

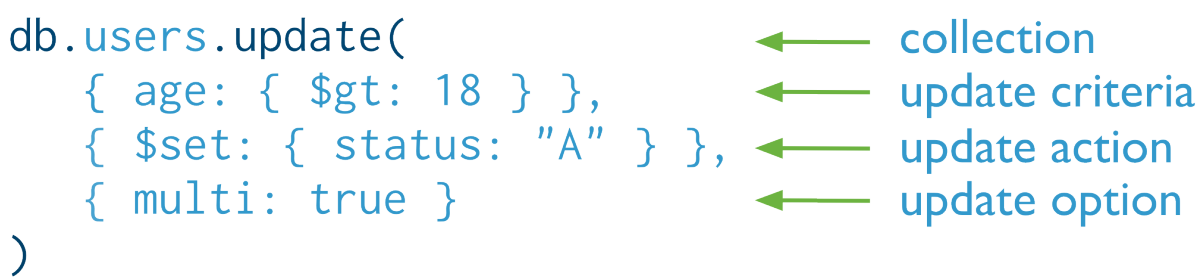
Other Methods to Add Documents You can also add new documents to a collection using methods that have an *upsert* (page 80) option. If the option is set to `true`, these methods will either modify existing documents or add a new document when no matching documents exist for the query. For more information, see *Update Behavior with the upsert Option* (page 80).

Update

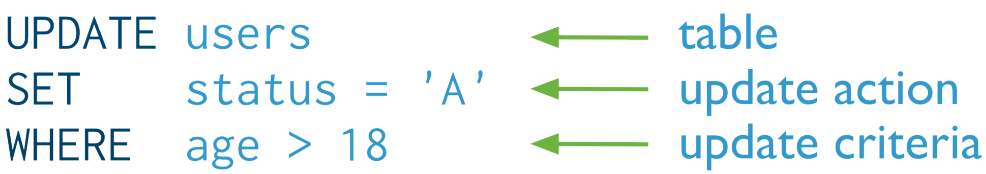
In MongoDB, the `db.collection.update()` method modifies existing *documents* in a *collection*. The `db.collection.update()` method can accept query criteria to determine which documents to update as well as an options document that affects its behavior, such as the `multi` option to update multiple documents.

Operations performed by an update are atomic within a single document. For example, you can safely use the `$inc` and `$mul` operators to modify frequently-changed fields in concurrent applications.

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



The following diagram shows the same query in SQL:



Example

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

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

For more information, see `db.collection.update()` and *update() Examples*.

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

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

When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk.

MongoDB preserves the order of the document fields following write operations *except* for the following cases:

- The `_id` field is always the first field in the document.
- Updates that include `renaming` of field names may result in the reordering of fields in the document.

Changed in version 2.6: Starting in version 2.6, MongoDB actively attempts to preserve the field order in a document. Before version 2.6, MongoDB did not actively preserve the order of the fields in a document.

Update Behavior with the `upsert` Option If the `update()` method includes *upsert: true and* no documents match the query portion of the update operation, then the update operation creates a new document. If there are matching documents, then the update operation with the *upsert: true* modifies the matching document or documents.

By specifying *upsert: true*, applications can indicate, in a *single* operation, that if no matching documents are found for the update, an insert should be performed. See `update()` for details on performing an *upsert*.

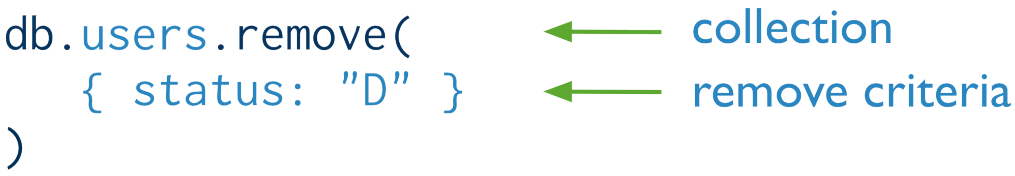
Changed in version 2.6: In 2.6, the new `Bulk()` methods and the underlying `update` command allow you to perform many updates with `upsert: true` operations in a single call.

If you create documents using the `upsert` option to `update()` consider using a *a unique index* to prevent duplicated operations.

Remove

In MongoDB, the `db.collection.remove()` method deletes documents from a collection. The `db.collection.remove()` method accepts a query criteria to determine which documents to remove.

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



The following diagram shows the same query in SQL:

Example

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

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

For more information, see `db.collection.remove()` method and *Remove Documents* (page 111).

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

Isolation of Write Operations

The modification of a single document is always atomic, even if the write operation modifies multiple embedded documents *within* that document. No other operations are atomic.

If a write operation modifies multiple documents, the operation as a whole is not atomic, and other operations may interleave. You can, however, attempt to isolate a write operation that affects multiple documents using the `isolation operator`.

For more information *Atomicity and Transactions* (page 86).

Additional Methods

The `db.collection.save()` method can either update an existing document or insert a document if the document cannot be found by the `_id` field. See `db.collection.save()` for more information and examples.

MongoDB also provides methods to perform write operations in bulk. See `Bulk()` for more information.

Write Concern

On this page

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

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

Changed in version 2.6: A new protocol for *write operations* (page 832) integrates write concern with the write operations.

For details on write concern configurations, see *Write Concern Reference* (page 135).

Considerations

Default Write Concern The `mongo` shell and the MongoDB drivers use *Acknowledged* (page 82) as the default write concern.

See *Acknowledged* (page 82) for more information, including when this write concern became the default.

Timeouts Clients can set a *wtimeout* (page 136) value as part of a *replica acknowledged* (page 83) write concern. If the write concern is not satisfied in the specified interval, the operation returns an error, even if the write concern will eventually succeed.

MongoDB does not "rollback" or undo modifications made before the `wtimeout` interval expired.

Write Concern Levels

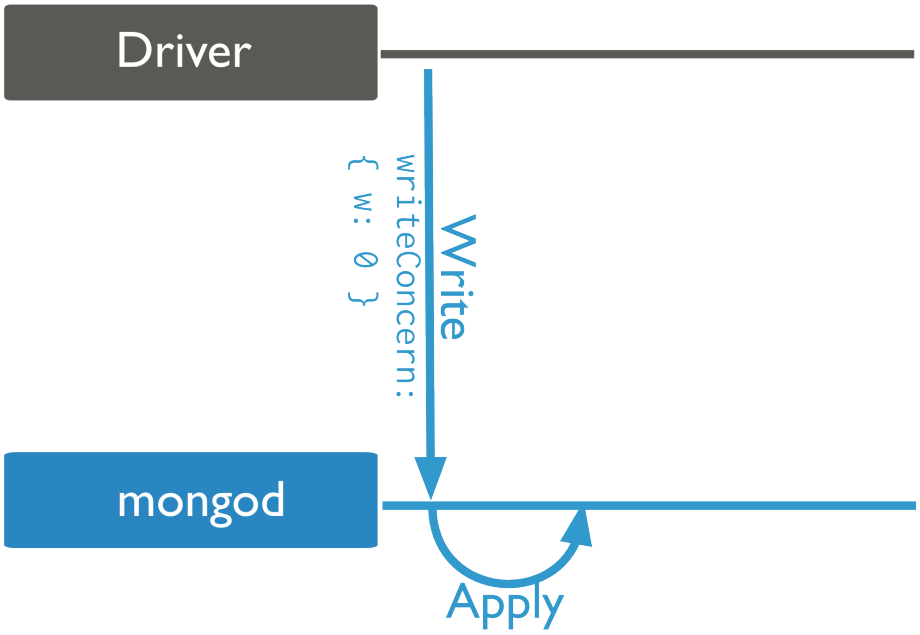
MongoDB has the following levels of conceptual write concern, listed from weakest to strongest:

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

Before the releases outlined in *Default Write Concern Change* (page 907), this was the default write concern.

Acknowledged With a receipt *acknowledged* write concern, the `mongod` confirms that it received the write operation and applied the change to the in-memory view of data. *Acknowledged* write concern allows clients to catch network, duplicate key, and other errors.

MongoDB uses the *acknowledged* write concern by default starting in the driver releases outlined in *Releases* (page 908).



Changed in version 2.6: The `mongo` shell write methods now incorporates the *write concern* (page 82) in the write methods and provide the default write concern whether run interactively or in a script. See *Write Method Acknowledgements* (page 838) for details.

Acknowledged write concern does *not* confirm that the write operation has persisted to the disk system.

Journaled With a *journaled* write concern, the MongoDB acknowledges the write operation only after committing the data to the *journal*. This write concern ensures that MongoDB can recover the data following a shutdown or power interruption.

You must have journaling enabled to use this write concern.

With a *journaled* write concern, write operations must wait for the next journal commit. To reduce latency for these operations, MongoDB also increases the frequency that it commits operations to the journal. See `commitIntervalMs` 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.

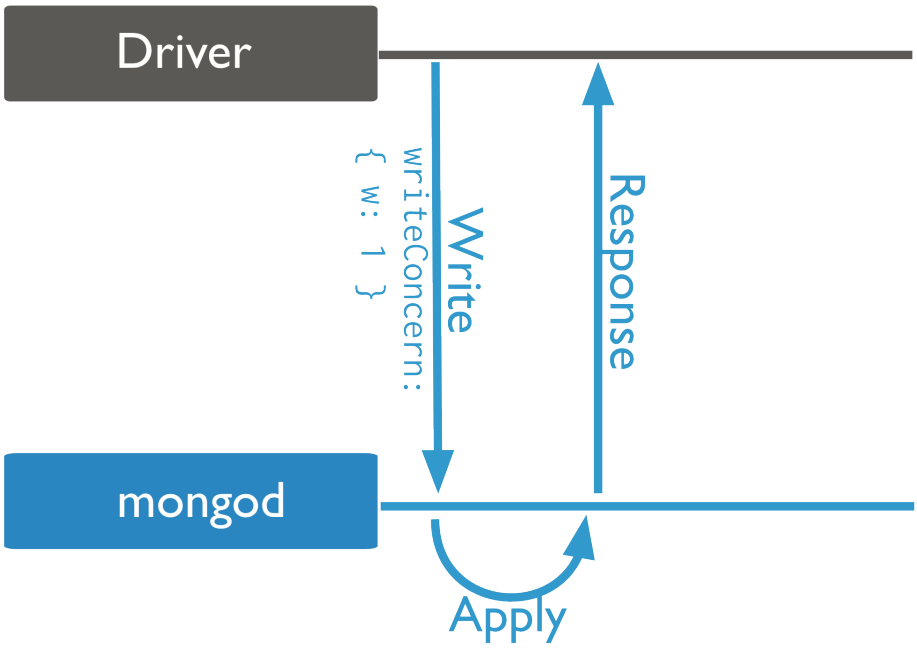
Replica Acknowledged *Replica sets* present additional considerations with regards to write concern. The default write concern only requires acknowledgement from the primary.

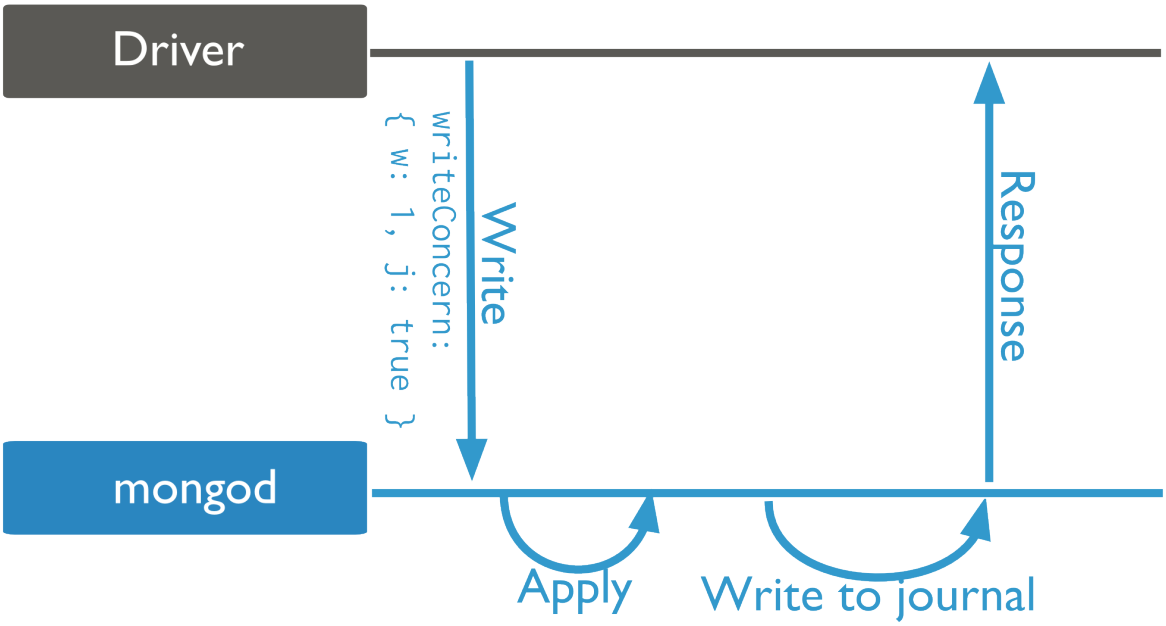
With *replica acknowledged* write concern, you can guarantee that the write operation propagates to additional members of the replica set. See *Write Concern for Replica Sets* (page 589) for more information.

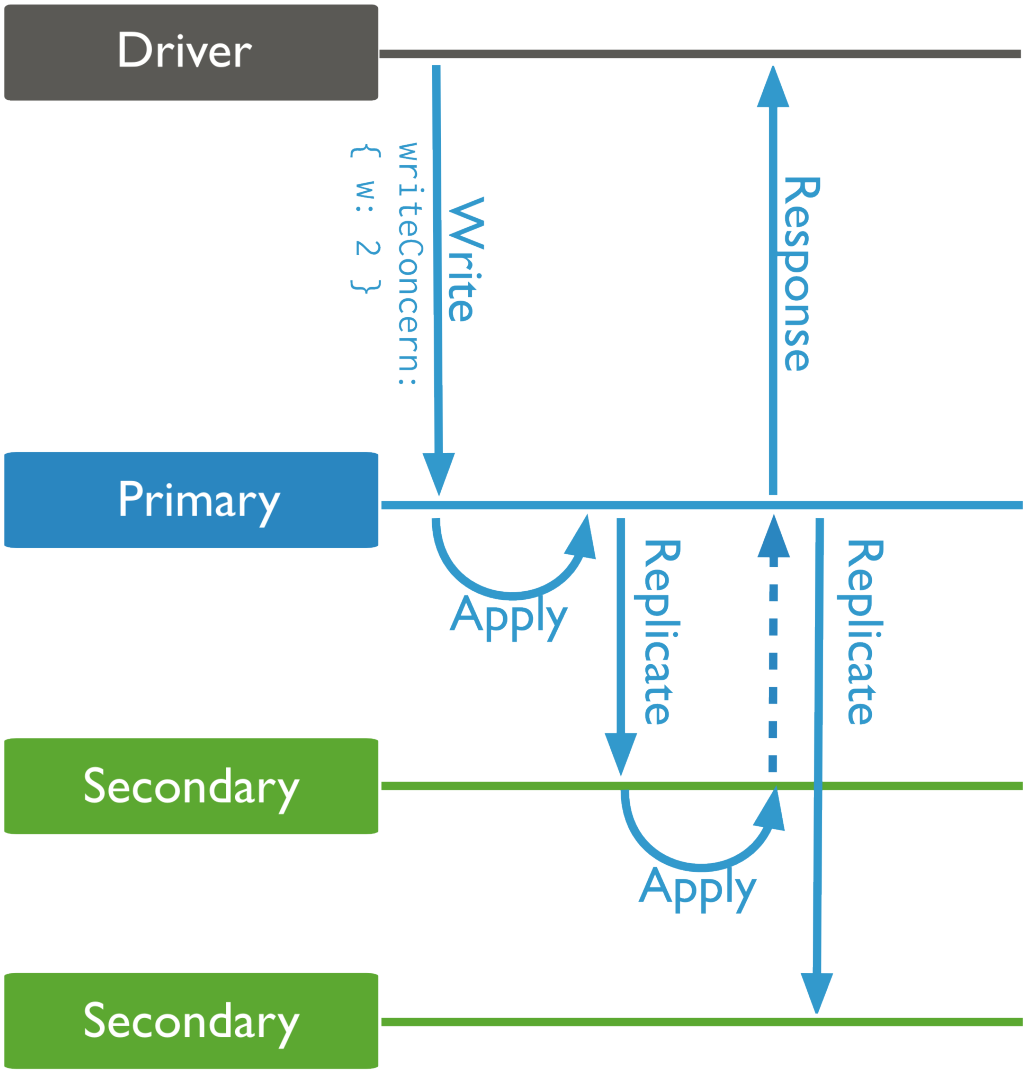
Note: Requiring *journaled* write concern in a replica set only requires a journal commit of the write operation to the *primary* of the set regardless of the level of *replica acknowledged* write concern.

See also:

Write Concern Reference (page 135)


Driver
writeConcern:
{ w: 1 }
Write
Response
mongod
Apply


Driver
writeConcern:
{ w: 1, j: true }
Write
Response
mongod
Apply
Write to journal


Driver
Write
writeConcern:
{ w: 2 }
Response
Primary
Apply
Replicate
Replicate
Secondary
Apply
Secondary

Atomicity and Transactions

On this page

In MongoDB, a write operation is atomic on the level of a single document, even if the operation modifies multiple embedded documents *within* a single document.

When a single write operation modifies multiple documents, the modification of each document is atomic, but the operation as a whole is not atomic and other operations may interleave. However, you can *isolate* a single write operation that affects multiple documents using the `$isolated` operator.

`$isolated` Operator

Using the `$isolated` operator, a write operation that affect multiple documents can prevent other processes from interleaving once the write operation modifies the first document. This ensures that no client sees the changes until the write operation completes or errors out.

Isolated write operation does not provide "all-or-nothing" atomicity. That is, an error during the write operation does not roll back all its changes that preceded the error.

The `$isolated` operator does **not** work on sharded clusters.

For an example of an update operation that uses the `$isolated` operator, see `$isolated`. For an example of a remove operation that uses the `$isolated` operator, see *isolate-remove-operations*.

Transaction-Like Semantics

Since a single document can contain multiple embedded documents, single-document atomicity is sufficient for many practical use cases. For cases where a sequence of write operations must operate as if in a single transaction, you can implement a *two-phase commit* (page 120) in your application.

However, two-phase commits can only offer transaction-*like* semantics. Using two-phase commit ensures data consistency, but it is possible for applications to return intermediate data during the two-phase commit or rollback.

For more information on two-phase commit and rollback, see *Perform Two Phase Commits* (page 120).

Concurrency Control

Concurrency control allows multiple applications to run concurrently without causing data inconsistency or conflicts.

An approach may be to create a *unique index* (page 506) on a field (or fields) that should have only unique values (or unique combination of values) prevents duplicate insertions or updates that result in duplicate values. For examples of use cases, see *update() and Unique Index* and *findAndModify() and Unique Index*.

Another approach is to specify the expected current value of a field in the query predicate for the write operations. For an example, see *Update if Current* (page 126).

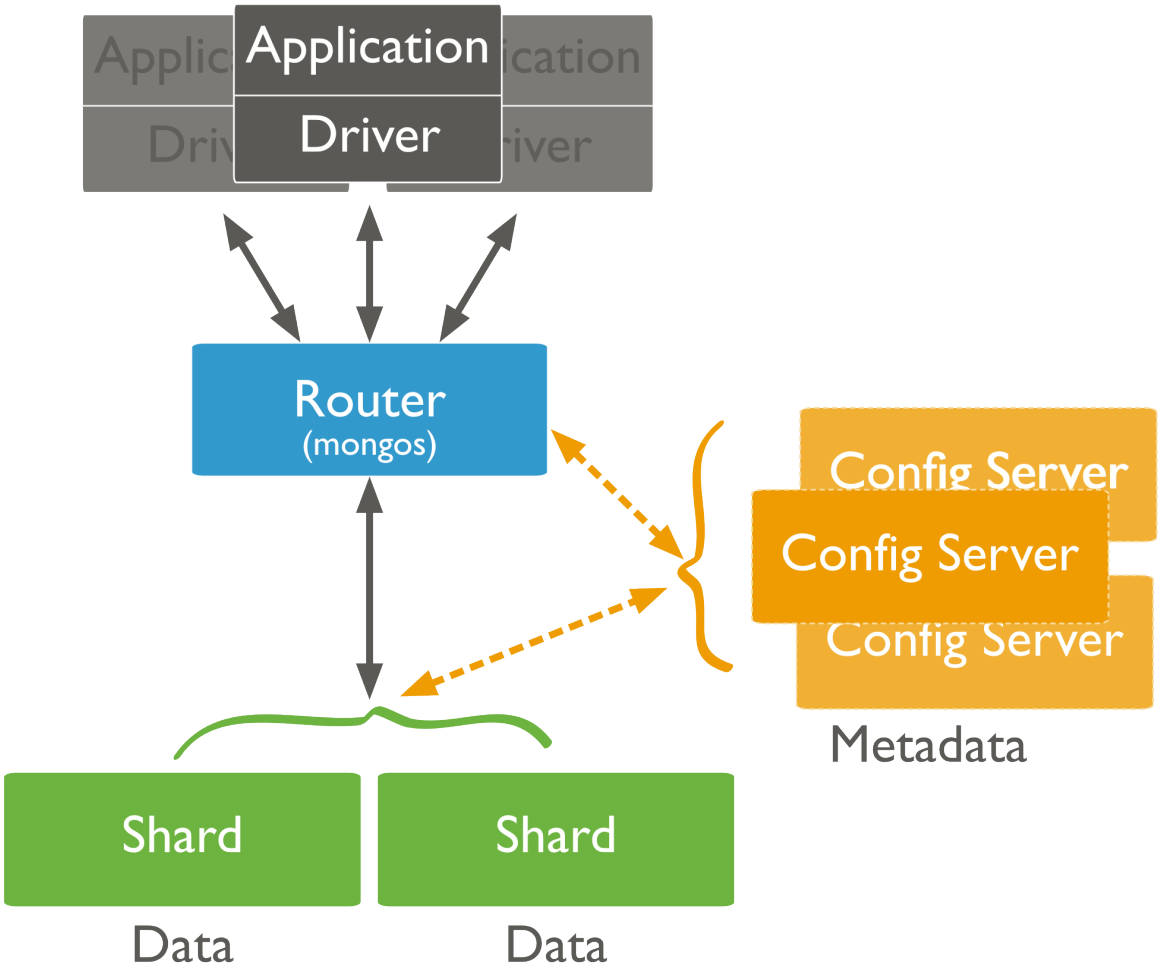
The two-phase commit pattern provides a variation where the query predicate includes the *application identifier* (page 124) as well as the expected state of the data in the write operation.

Distributed Write Operations

On this page

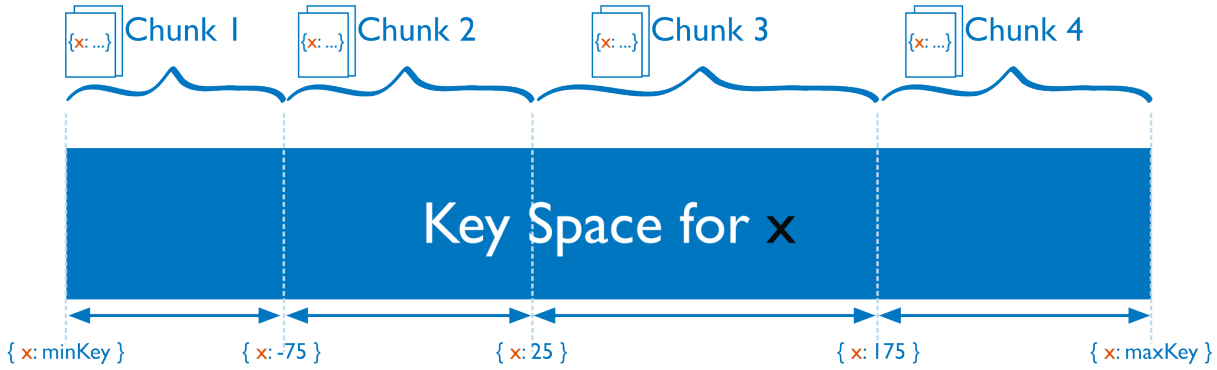
Write Operations on Sharded Clusters

For sharded collections in a *sharded cluster*, the `mongos` directs write operations from applications to the shards that are responsible for the specific *portion* of the data set. The `mongos` uses the cluster metadata from the *config database* (page 684) to route the write operation to the appropriate shards.



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

Important: Update operations that affect a *single* document **must** include the *shard key* or the `_id` field. Updates



that affect multiple documents are more efficient in some situations if they have the *shard key*, but can be broadcast to all shards.

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

For more information, see *Sharded Cluster Tutorials* (page 704) and *Bulk Write Operations* (page 92).

Write Operations on Replica Sets

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

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 583) in the form of *rollbacks* (page 587) as well as general *read consistency* (page 588).

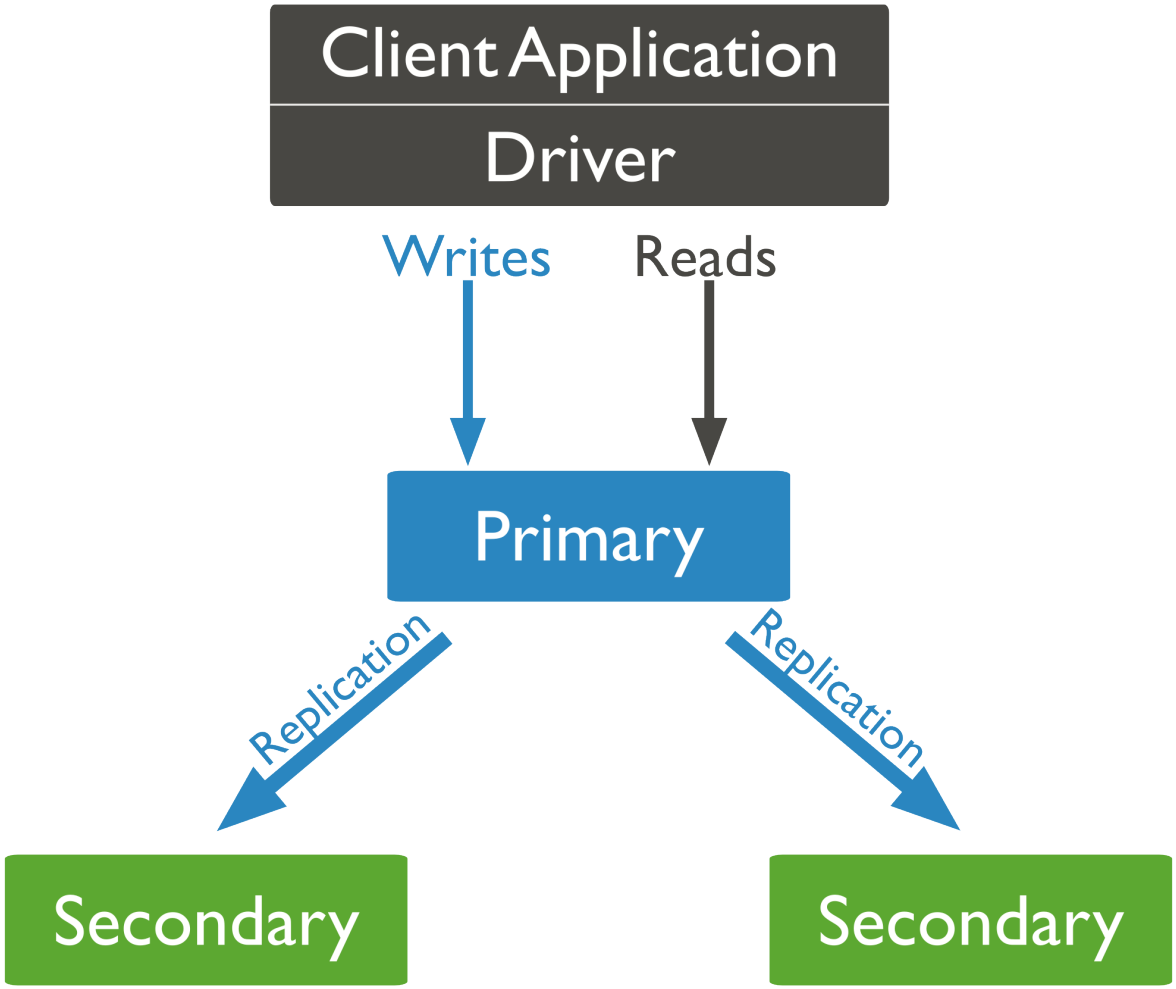
To help avoid this issue, you can customize the *write concern* (page 82) to return confirmation of the write operation to another member [5] of the replica set every 100 or 1,000 operations. This provides an opportunity for secondaries to catch up with the primary. Write concern can slow the overall progress of write operations but ensure that the secondaries can maintain a largely current state with respect to the primary.

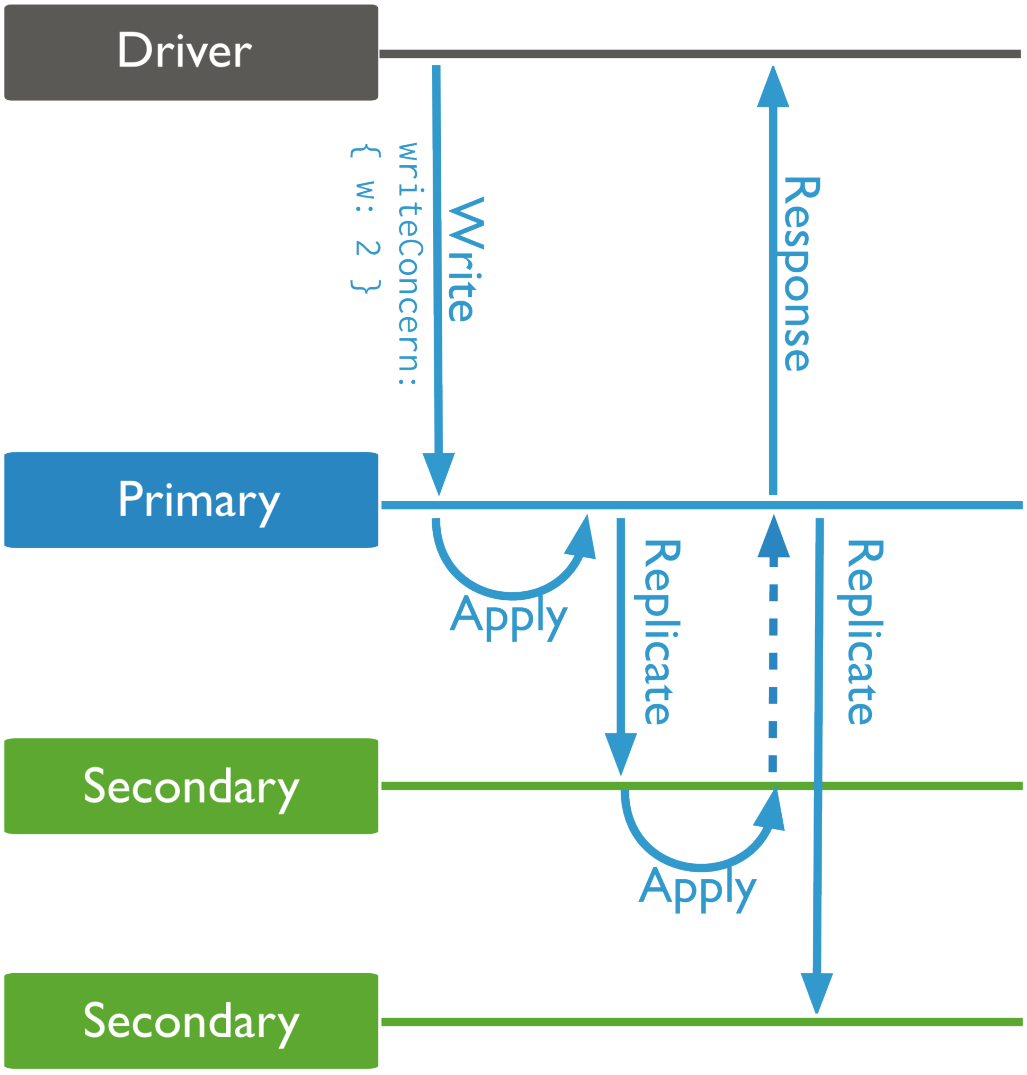
For more information on replica sets and write operations, see *Replica Acknowledged* (page 83), *Oplog Size* (page 597), and *Change the Size of the Oplog* (page 634).

Write Operation Performance

[5] Intermittently issuing a write concern 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.

Changed in version 2.6: In *Master/Slave* (page 600) deployments, MongoDB treats `w: "majority"` as equivalent to `w: 1`. In earlier versions of MongoDB, `w: "majority"` produces an error in *master/slave* (page 600) deployments.


Client Application
Driver
Writes
Reads
Primary
Replication
Replication
Secondary
Secondary


Driver
Write
writeConcern:
{ w: 2 }
Response
Primary
Apply
Replicate
Replicate
Secondary
Apply
Secondary

On this page

Indexes

After every insert, update, or delete operation, MongoDB must update *every* index associated with the collection in addition to the data itself. Therefore, every index on a collection adds some amount of overhead for the performance of write operations. [6]

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

For indexes and queries, see *Query Optimization* (page 70). For more information on indexes, see *Indexes* (page 481) and *Indexing Strategies* (page 551).

Document Growth

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

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

In-place-updates are significantly more efficient than updates that cause document growth. When possible, use *data models* (page 151) that minimize the need for document growth.

See *Storage* (page 94) for more information.

Storage Performance

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

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

See

Production Notes (page 210) for recommendations regarding additional hardware and configuration options.

Journaling MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 77) durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal.

[6] For inserts and updates to un-indexed fields, the overhead for *sparse indexes* (page 507) is less than for non-sparse indexes. Also for non-sparse indexes, updates that do not change the record size have less indexing overhead.

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

- if the journal and the data file reside on the same block device, the data files and the journal may have to contend for a finite number of available write operations. Moving the journal to a separate device may increase the capacity for write operations.
- if applications specify *write concern* (page 82) that includes *journaled* (page 83), `mongod` will decrease the duration between journal commits, which can increases the overall write load.
- the duration between journal commits is configurable using the `commitIntervalMs` run-time option. Decreasing the period between journal commits will increase the number of write operations, which can limit MongoDB's capacity for write operations. Increasing the amount of time between commits may decrease the total number of write operation, but also increases the chance that the journal will not record a write operation in the event of a failure.

For additional information on journaling, see *Journaling Mechanics* (page 309).

Bulk Write Operations

On this page

Overview

MongoDB provides clients the ability to perform write operations in bulk. Bulk write operations affect a *single* collection. MongoDB allows applications to determine the acceptable level of acknowledgement required for bulk write operations.

New `Bulk` methods provide the ability to perform bulk insert, update, and remove operations. MongoDB also supports bulk insert through passing an array of documents to the `db.collection.insert()` method.

Changed in version 2.6: Previous versions of MongoDB provided the ability for bulk inserts only. With previous versions, clients could perform bulk inserts by passing an array of documents to the db.collection.insert()[7] method. To see the documentation for earlier versions, see Bulk Inserts[8].

Ordered vs Unordered Operations

Bulk write operations can be either *ordered* or *unordered*. With an ordered list of operations, MongoDB executes the operations serially. If an error occurs during the processing of one of the write operations, MongoDB will return without processing any remaining write operations in the list.

With an unordered list of operations, MongoDB can execute the operations in parallel. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

Executing an ordered list of operations on a sharded collection will generally be slower than executing an unordered list since with an ordered list, each operation must wait for the previous operation to finish.

[7] http://docs.mongodb.org/v2.4/core/bulk-inserts
[8] http://docs.mongodb.org/v2.4/core/bulk-inserts

`Bulk` Methods

To use the `Bulk()` methods:

1. Initialize a list of operations using either `db.collection.initializeUnorderedBulkOp()` or `db.collection.initializeOrderedBulkOp()`.
2. Add write operations to the list using the following methods:
   - `Bulk.insert()`
   - `Bulk.find()`
   - `Bulk.find.upsert()`
   - `Bulk.find.update()`
   - `Bulk.find.updateOne()`
   - `Bulk.find.replaceOne()`
   - `Bulk.find.remove()`
   - `Bulk.find.removeOne()`
3. To execute the list of operations, use the `Bulk.execute()` method. You can specify the write concern for the list in the `Bulk.execute()` method.

   Once executed, you cannot re-execute the list without reinitializing.

For example,

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { _id: 1, item: "abc123", status: "A", soldQty: 5000 } );
bulk.insert( { _id: 2, item: "abc456", status: "A", soldQty: 150 } );
bulk.insert( { _id: 3, item: "abc789", status: "P", soldQty: 0 } );
bulk.execute( { w: "majority", wtimeout: 5000 } );
```

For more examples, refer to the reference page for each `http://docs.mongodb.org/manual/reference/method/js-bul` method. For information and examples on performing bulk insert using the `db.collection.insert()`, see `db.collection.insert()`.

See also:

New Write Operation Protocol (page 832)

`Bulk` Execution Mechanics

When executing an `ordered` list of operations, MongoDB groups adjacent operations by the `operation type`. When executing an `unordered` list of operations, MongoDB groups and may also reorder the operations to increase performance. As such, when performing *unordered* bulk operations, applications should not depend on the ordering.

Each group of operations can have at most `1000 operations`. If a group exceeds this `limit`, MongoDB will divide the group into smaller groups of 1000 or less. For example, if the bulk operations list consists of 2000 insert operations, MongoDB creates 2 groups, each with 1000 operations.

The sizes and grouping mechanics are internal performance details and are subject to change in future versions.

To see how the operations are grouped for a bulk operation execution, call `Bulk.getOperations()` *after* the execution.

For more information, see `Bulk.execute()`.

Strategies for Bulk Inserts to a Sharded Collection

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

Pre-Split the Collection If the sharded collection is empty, then the collection has only one initial *chunk*, which resides on a single shard. MongoDB must then take time to receive data, create splits, and distribute the split chunks to the available shards. To avoid this performance cost, you can pre-split the collection, as described in *Split Chunks in a Sharded Cluster* (page 738).

Insert to Multiple `mongos` To parallelize import processes, send bulk insert or insert operations to more than one `mongos` instance. For *empty* collections, first pre-split the collection as described in *Split Chunks in a Sharded Cluster* (page 738).

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

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

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

Example

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

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

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

See also:

Shard Keys (page 687) for information on choosing a sharded key. Also see *Shard Key Internals* (page 687) (in particular, *Choosing a Shard Key* (page 709)).

Storage

On this page

Data Model

MongoDB stores data in the form of *BSON* documents, which are rich mappings of keys, or field names, to values. BSON supports a rich collection of types, and fields in BSON documents may hold arrays of values or embedded documents. All documents in MongoDB must be less than 16MB, which is the `BSON document size`.

Every document in MongoDB is stored in a *record* which contains the document itself and extra space, or *padding*, which allows the document to grow as the result of updates.

All records are contiguously located on disk, and when a document becomes larger than the allocated record, MongoDB must allocate a new record. New allocations require MongoDB to move a document and update all indexes that refer to the document, which takes more time than in-place updates and leads to storage fragmentation.

All records are part of a *collection*, which is a logical grouping of documents in a MongoDB database. The documents in a collection share a set of indexes, and typically these documents share common fields and structure.

In MongoDB the *database* construct is a group of related collections. Each database has a distinct set of data files and can contain a large number of collections. Also, each database has one distinct write lock, that blocks operations to the database during write operations. A single MongoDB deployment may have many databases.

Journal

In order to ensure that all modifications to a MongoDB data set are durably written to disk, MongoDB records all modifications to a journal that it writes to disk more frequently than it writes the data files. The journal allows MongoDB to successfully recover data from data files after a `mongod` instance exits without flushing all changes.

See *Journaling Mechanics* (page 309) for more information about the journal in MongoDB.

Record Allocation Strategies

MongoDB supports multiple record allocation strategies that determine how `mongod` adds padding to a document when creating a record. Because documents in MongoDB may grow after insertion and all records are contiguous on disk, the padding can reduce the need to relocate documents on disk following updates. Relocations are less efficient than in-place updates, and can lead to storage fragmentation. As a result, all padding strategies trade additional space for increased efficiency and decreased fragmentation.

Different allocation strategies support different kinds of workloads: the *power of 2 allocations* (page 95) are more efficient for insert/update/delete workloads; while *exact fit allocations* (page 96) is ideal for collections *without* update and delete workloads.

Power of 2 Sized Allocations Changed in version 2.6: For all new collections, `usePowerOf2Sizes` became the default allocation strategy. To change the default allocation strategy, use the `newCollectionsUsePowerOf2Sizes` parameter.

mongod uses an allocation strategy called usePowerOf2Sizes where each record has a size in bytes that is a power of 2 (e.g. 32, 64, 128, 256, 512...16777216.) The smallest allocation for a document is 32 bytes. The power of 2 sizes allocation strategy has two key properties:

- there are a limited number of record allocation sizes, which makes it easier for mongod to reuse existing allocations, which will reduce fragmentation in some cases.
- in many cases, the record allocations are significantly larger than the documents they hold. This allows documents to grow while minimizing or eliminating the chance that the mongod will need to allocate a new record if the document grows.

The usePowerOf2Sizes strategy does not *eliminate* document reallocation as a result of document growth, but it minimizes its occurrence in many common operations.

Exact Fit Allocation The exact fit allocation strategy allocates record sizes based on the size of the document and an additional *padding factor*. Each collection has its own padding factor, which defaults to 1 when you insert the first document in a collection. MongoDB dynamically adjusts the padding factor up to 2 depending on the rate of growth of the documents over the life of the collection.

To estimate total record size, compute the product of the padding factor and the size of the document. That is:

```
record size = paddingFactor * <document size>
```

The size of each record in a collection reflects the size of the padding factor at the time of allocation. See the paddingFactor field in the output of db.collection.stats() to see the current padding factor for a collection.

On average, this exact fit allocation strategy uses less storage space than the usePowerOf2Sizes strategy but will result in higher levels of storage fragmentation if documents grow beyond the size of their initial allocation.

The compact and repairDatabase operations remove padding by default, as do the mongodump and mongorestore. compact does allow you to specify a padding for records during compaction.

Capped Collections

Capped collections are fixed-size collections that support high-throughput operations that store records in insertion order. Capped collections work like circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

See *Capped Collections* (page 219) for more information.

3.3 MongoDB CRUD Tutorials

The following tutorials provide instructions for querying and modifying data. For a higher-level overview of these operations, see *MongoDB CRUD Operations* (page 61).

Iterate a Cursor in the mongo Shell **(page 115)** Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

Analyze Query Performance **(page 117)** Use query introspection (i.e. `explain`) to analyze the efficiency of queries and determine how a query uses available indexes.

Perform Two Phase Commits **(page 120)** Use two-phase commits when writing data to multiple documents.

Update Document if Current **(page 126)** Update a document only if it has not changed since it was last read.

Create Tailable Cursor **(page 128)** Create tailable cursors for use in capped collections with high numbers of write operations for which an index would be too expensive.

Create an Auto-Incrementing Sequence Field **(page 130)** Describes how to create an incrementing sequence number for the `_id` field using a Counters Collection or an Optimistic Loop.

3.3.1 Insert Documents

On this page

In MongoDB, the `db.collection.insert()` method adds new documents into a collection.

Insert a Document

Step 1: Insert a document into a collection.

Insert a document into a collection named `inventory`. The operation will create the collection if the collection does not currently exist.

```
db.inventory.insert(
   {
     item: "ABC1",
     details: {
        model: "14Q3",
        manufacturer: "XYZ Company"
     },
     stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
     category: "clothing"
   }
)
```

The operation returns a `WriteResult` object with the status of the operation. A successful insert of the document returns the following object:

```
WriteResult({ "nInserted" : 1 })
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `WriteResult` object will contain the error information.

Step 2: Review the inserted document.

If the insert operation is successful, verify the insertion by querying the collection.

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

The document you inserted should return.

```
{ "_id" : ObjectId("53d98f133bb604791249ca99"), "item" : "ABC1", "details" : { "model" : "14Q3", "mar
```

The returned document shows that MongoDB added an `_id` field to the document. If a client inserts a document that does not contain the `_id` field, MongoDB adds the field with the value set to a generated ObjectId[9]. The ObjectId[10] values in your documents will differ from the ones shown.

Insert an Array of Documents

You can pass an array of documents to the `db.collection.insert()` method to insert multiple documents.

Step 1: Create an array of documents.

Define a variable `mydocuments` that holds an array of documents to insert.

```
var mydocuments =
    [
      {
        item: "ABC2",
        details: { model: "14Q3", manufacturer: "M1 Corporation" },
        stock: [ { size: "M", qty: 50 } ],
        category: "clothing"
      },
      {
        item: "MNO2",
        details: { model: "14Q3", manufacturer: "ABC Company" },
        stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
        category: "clothing"
      },
      {
        item: "IJK2",
        details: { model: "14Q2", manufacturer: "M5 Corporation" },
        stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
        category: "houseware"
      }
    ];
```

Step 2: Insert the documents.

Pass the `mydocuments` array to the `db.collection.insert()` to perform a bulk insert.

```
db.inventory.insert( mydocuments );
```

The method returns a `BulkWriteResult` object with the status of the operation. A successful insert of the documents returns the following object:

[9] https://docs.mongodb.org/manual/reference/object-id
[10] https://docs.mongodb.org/manual/reference/object-id

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 3,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `BulkWriteResult` object will contain information regarding the error.

The inserted documents will each have an `_id` field added by MongoDB.

Insert Multiple Documents with `Bulk`

New in version 2.6.

MongoDB provides a `Bulk()` API that you can use to perform multiple write operations in bulk. The following sequence of operations describes how you would use the `Bulk()` API to insert a group of documents into a MongoDB collection.

Step 1: Initialize a `Bulk` operations builder.

Initialize a `Bulk` operations builder for the collection `inventory`.

```
var bulk = db.inventory.initializeUnorderedBulkOp();
```

The operation returns an unordered operations builder which maintains a list of operations to perform. Unordered operations means that MongoDB can execute in parallel as well as in nondeterministic order. If an error occurs during the processing of one of the write operations, MongoDB will continue to process remaining write operations in the list.

You can also initialize an ordered operations builder; see `db.collection.initializeOrderedBulkOp()` for details.

Step 2: Add insert operations to the `bulk` object.

Add two insert operations to the `bulk` object using the `Bulk.insert()` method.

```
bulk.insert(
   {
     item: "BE10",
     details: { model: "14Q2", manufacturer: "XYZ Company" },
     stock: [ { size: "L", qty: 5 } ],
     category: "clothing"
   }
);
bulk.insert(
   {
     item: "ZYT1",
     details: { model: "14Q1", manufacturer: "ABC Company"  },
     stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 } ],
```

```
      category: "houseware"
   }
);
```

Step 3: Execute the bulk operation.

Call the `execute()` method on the `bulk` object to execute the operations in its list.

```
bulk.execute();
```

The method returns a `BulkWriteResult` object with the status of the operation. A successful insert of the documents returns the following object:

```
BulkWriteResult({
   "writeErrors" : [ ],
   "writeConcernErrors" : [ ],
   "nInserted" : 2,
   "nUpserted" : 0,
   "nMatched" : 0,
   "nModified" : 0,
   "nRemoved" : 0,
   "upserted" : [ ]
})
```

The `nInserted` field specifies the number of documents inserted. If the operation encounters an error, the `BulkWriteResult` object will contain information regarding the error.

Additional Examples and Methods

For more examples, see `db.collection.insert()`.

The `db.collection.update()` method, the `db.collection.findAndModify()`, and the `db.collection.save()` method can also add new documents. See the individual reference pages for the methods for more information and examples.

3.3.2 Query Documents

On this page

In MongoDB, the `db.collection.find()` method retrieves documents from a collection. [11] The `db.collection.find()` method returns a *cursor* (page 68) to the retrieved documents.

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

This tutorial provides examples of read operations using the `db.collection.find()` method in the `mongo` shell. In these examples, the retrieved documents contain all their fields. To restrict the fields to return in the retrieved documents, see *Limit Fields to Return from a Query* (page 112).

Select All Documents in a Collection

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

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

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

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

Specify Equality Condition

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

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

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

Specify Conditions Using Query Operators

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

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

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

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

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

Specify AND Conditions

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

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

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

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

Specify OR Conditions

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

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

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

Specify AND as well as OR Conditions

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

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

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

Embedded Documents

When the field holds an embedded document, a query can either specify an exact match on the embedded document or specify a match by individual fields in the embedded document using the *dot notation*.

Exact Match on the Embedded Document

To specify an equality match on the whole embedded document, use the query document `{ <field>: <value> }` where `<value>` is the document to match. Equality matches on an embedded document require an *exact* match of the specified `<value>`, including the field order.

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

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

Equality Match on Fields within an Embedded Document

Use the *dot notation* to match by specific fields in an embedded document. Equality matches for specific fields in an embedded document will select documents in the collection where the embedded document contains the specified fields with the specified values. The embedded document can contain additional fields.

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

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

Arrays

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

If you specify multiple conditions using the `$elemMatch` operator, the array must contain at least one element that satisfies all the conditions. See *Single Element Satisfies the Criteria* (page 104).

If you specify multiple conditions without using the `$elemMatch` operator, then some combination of the array elements, not necessarily a single element, must satisfy all the conditions; i.e. different elements in the array can satisfy different parts of the conditions. See *Combination of Elements Satisfies the Criteria* (page 104).

Consider an `inventory` collection that contains the following documents:

```
{ _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] }
{ _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] }
{ _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] }
```

Exact Match on an Array

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

The following example queries for all documents where the field `ratings` is an array that holds exactly three elements, `5`, `8`, and `9`, in this order:

```
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```

The operation returns the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Match an Array Element

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

The following example queries for all documents where `ratings` is an array that contains `5` as one of its elements:

```
db.inventory.find( { ratings: 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Match a Specific Element of an Array

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

In the following example, the query uses the *dot notation* to match all documents where the `ratings` array contains `5` as the first element:

```
db.inventory.find( { 'ratings.0': 5 } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
```

Specify Multiple Criteria for Array Elements

Single Element Satisfies the Criteria Use `$elemMatch` operator to specify multiple criteria on the elements of an array such that at least one array element satisfies all the specified criteria.

The following example queries for documents where the `ratings` array contains at least one element that is greater than (`$gt`) `5` and less than (`$lt`) `9`:

```
db.inventory.find( { ratings: { $elemMatch: { $gt: 5, $lt: 9 } } } )
```

The operation returns the following documents, whose `ratings` array contains the element `8` which meets the criteria:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

Combination of Elements Satisfies the Criteria The following example queries for documents where the `ratings` array contains elements that in some combination satisfy the query conditions; e.g., one element can satisfy the greater than `5` condition and another element can satisfy the less than `9` condition, or a single element can satisfy both:

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```

The operation returns the following documents:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
{ "_id" : 6, "type" : "food", "item" : "bbb", "ratings" : [ 5, 9 ] }
{ "_id" : 7, "type" : "food", "item" : "ccc", "ratings" : [ 9, 5, 8 ] }
```

The document with the `"ratings" : [ 5, 9 ]` matches the query since the element `9` is greater than `5` (the first condition) and the element `5` is less than `9` (the second condition).

Array of Embedded Documents

Consider that the `inventory` collection includes the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}

{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Match a Field in the Embedded Document Using the Array Index If you know the array index of the embedded document, you can specify the document using the embedded document's position using the *dot notation*.

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

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

The operation returns the following document:

```
{
   _id: 100,
   type: "food",
   item: "xyz",
   qty: 25,
   price: 2.5,
   ratings: [ 5, 8, 9 ],
   memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
```

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

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

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

The operation returns the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
```

```
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]
}
```

Specify Multiple Criteria for Array of Documents

Single Element Satisfies the Criteria Use `$elemMatch` operator to specify multiple criteria on an array of embedded documents such that at least one embedded document satisfies all the specified criteria.

The following example queries for documents where the `memos` array has at least one embedded document that contains both the field `memo` equal to `'on time'` and the field `by` equal to `'shipping'`:

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

The operation returns the following document:

```
{
   _id: 100,
   type: "food",
   item: "xyz",
   qty: 25,
   price: 2.5,
   ratings: [ 5, 8, 9 ],
   memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
```

Combination of Elements Satisfies the Criteria The following example queries for documents where the `memos` array contains elements that in some combination satisfy the query conditions; e.g. one element satisfies the field `memo` equal to `'on time'` condition and another element satisfies the field `by` equal to `'shipping'` condition, or a single element can satisfy both criteria:

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

```
  }
)
```

The query returns the following documents:

```
{
  _id: 100,
  type: "food",
  item: "xyz",
  qty: 25,
  price: 2.5,
  ratings: [ 5, 8, 9 ],
  memos: [ { memo: "on time", by: "shipping" }, { memo: "approved", by: "billing" } ]
}
{
  _id: 101,
  type: "fruit",
  item: "jkl",
  qty: 10,
  price: 4.25,
  ratings: [ 5, 9 ],
  memos: [ { memo: "on time", by: "payment" }, { memo: "delayed", by: "shipping" } ]

}
```

See also:

Limit Fields to Return from a Query (page 112)

3.3.3 Modify Documents

On this page

MongoDB provides the `update()` method to update the documents of a collection. The method accepts as its parameters:

- an update conditions document to match the documents to update,
- an update operations document to specify the modification to perform, and
- an options document.

To specify the update condition, use the same structure and syntax as the query conditions.

By default, `update()` updates a single document. To update multiple documents, use the *multi* option.

Update Specific Fields in a Document

To change a field value, MongoDB provides update operators[12], such as `$set` to modify values.

[12] https://docs.mongodb.org/manual/reference/operator/update

Some update operators, such as `$set`, will create the field if the field does not exist. See the individual update operator[13] reference.

Step 1: Use update operators to change field values.

For the document with `item` equal to `"MNO2"`, use the `$set` operator to update the `category` field and the `details` field to the specified values and the `$currentDate` operator to update the field `lastModified` with the current date.

```
db.inventory.update(
    { item: "MNO2" },
    {
      $set: {
        category: "apparel",
        details: { model: "14Q3", manufacturer: "XYZ Company" }
      },
      $currentDate: { lastModified: true }
    }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The `nMatched` field specifies the number of existing documents matched for the update, and `nModified` specifies the number of existing documents modified.

Step 2: Update an embedded field.

To update a field within an embedded document, use the *dot notation*. When using the dot notation, enclose the whole dotted field name in quotes.

The following updates the `model` field within the embedded `details` document.

```
db.inventory.update(
  { item: "ABC1" },
  { $set: { "details.model": "14Q2" } }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Step 3: Update multiple documents.

By default, the `update()` method updates a single document. To update multiple documents, use the `multi` option in the `update()` method.

Update the `category` field to `"apparel"` and update the `lastModified` field to the current date for *all* documents that have `category` field equal to `"clothing"`.

[13] https://docs.mongodb.org/manual/reference/operator/update

```
db.inventory.update(
   { category: "clothing" },
   {
     $set: { category: "apparel" },
     $currentDate: { lastModified: true }
   },
   { multi: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 3 })
```

Replace the Document

To replace the entire content of a document except for the `_id` field, pass an entirely new document as the second argument to `update()`.

The replacement document can have different fields from the original document. In the replacement document, you can omit the `_id` field since the `_id` field is immutable. If you do include the `_id` field, it must be the same value as the existing value.

Step 1: Replace a document.

The following operation replaces the document with `item` equal to `"BE10"`. The newly replaced document will only contain the the `_id` field and the fields in the replacement document.

```
db.inventory.update(
   { item: "BE10" },
   {
     item: "BE05",
     stock: [ { size: "S", qty: 20 }, { size: "M", qty: 5 } ],
     category: "apparel"
   }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation. A successful update of the document returns the following object:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

`upsert` Option

By default, if no document matches the update query, the `update()` method does nothing.

However, by specifying *upsert: true*, the `update()` method either updates matching document or documents, or inserts a new document using the update specification if no matching document exists.

Step 1: Specify `upsert: true` for the update replacement operation.

When you specify `upsert: true` for an update operation to replace a document and no matching documents are found, MongoDB creates a new document using the equality conditions in the update conditions document, and replaces this document, except for the `_id` field if specified, with the update document.

The following operation either updates a matching document by replacing it with a new document or adds a new document if no matching document exists.

```
db.inventory.update(
   { item: "TBD1" },
   {
     item: "TBD1",
     details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
     stock: [ { "size" : "S", "qty" : 25 } ],
     category: "houseware"
   },
   { upsert: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation, including whether the `db.collection.update()` method modified an existing document or added a new document.

```
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
    "_id" : ObjectId("53dbd684babeaec6342ed6c7")
})
```

The `nMatched` field shows that the operation matched `0` documents.

The `nUpserted` of `1` shows that the update added a document.

The `nModified` of `0` specifies that no existing documents were updated.

The `_id` field shows the generated `_id` field for the added document.

Step 2: Specify `upsert: true` for the update specific fields operation.

When you specify `upsert: true` for an update operation that modifies specific fields and no matching documents are found, MongoDB creates a new document using the equality conditions in the update conditions document, and applies the modification as specified in the update document.

The following update operation either updates specific fields of a matching document or adds a new document if no matching document exists.

```
db.inventory.update(
   { item: "TBD2" },
   {
     $set: {
        details: { "model" : "14Q3", "manufacturer" : "IJK Co." },
        category: "houseware"
     }
   },
   { upsert: true }
)
```

The update operation returns a `WriteResult` object which contains the status of the operation, including whether the `db.collection.update()` method modified an existing document or added a new document.

```
WriteResult({
    "nMatched" : 0,
    "nUpserted" : 1,
    "nModified" : 0,
```

```
    "_id" : ObjectId("53dbd7c8babeaec6342ed6c8")
})
```

The `nMatched` field shows that the operation matched `0` documents.

The `nUpserted` of `1` shows that the update added a document.

The `nModified` of `0` specifies that no existing documents were updated.

The `_id` field shows the generated `_id` field for the added document.

Additional Examples and Methods

For more examples, see *Update examples* in the `db.collection.update()` reference page.

The `db.collection.findAndModify()` and the `db.collection.save()` method can also modify existing documents or insert a new one. See the individual reference pages for the methods for more information and examples.

3.3.4 Remove Documents

On this page

In MongoDB, the `db.collection.remove()` method removes documents from a collection. You can remove all documents from a collection, remove all documents that match a condition, or limit the operation to remove just a single document.

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

Remove All Documents

To remove all documents from a collection, pass an empty query document `{}` to the `remove()` method. The `remove()` method does not remove the indexes.

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

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

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

Remove Documents that Match a Condition

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

The following example removes all documents from the `inventory` collection where the `type` field equals `food`:

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

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

Remove a Single Document that Matches a Condition

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

The following example removes one document from the `inventory` collection where the `type` field equals `food`:

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

To delete a single document sorted by some specified order, use the *findAndModify()* method.

3.3.5 Limit Fields to Return from a Query

On this page

The *projection* document limits the fields to return for all matching documents. The projection document can specify the inclusion of fields or the exclusion of fields.

The specifications have the following forms:

Syntax	Description
`<field>: <1 or true>`	Specify the inclusion of a field.
`<field>: <0 or false>`	Specify the suppression of the field.

Important: The `_id` field is, by default, included in the result set. To suppress the `_id` field from the result set, specify `_id: 0` in the projection document.

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

This tutorial offers various query examples that limit the fields to return for all matching documents. The examples in this tutorial use a collection `inventory` and use the `db.collection.find()` method in the `mongo` shell. The `db.collection.find()` method returns a *cursor* (page 68) to the retrieved documents. For examples on query selection criteria, see *Query Documents* (page 100).

Return All Fields in Matching Documents

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

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

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

Return the Specified Fields and the _id Field Only

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

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

Return Specified Fields Only

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

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

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

Return All But the Excluded Field

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

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

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

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

Return Specific Fields in Embedded Documents

Use the *dot notation* (page 179) to return specific fields inside an embedded document. For example, the `inventory` collection contains the following document:

```
{
  "_id" : 3,
  "type" : "food",
  "item" : "aaa",
  "classification": { dept: "grocery", category: "chocolate"  }
}
```

The following operation returns all documents that match the query. The specified projection returns only the `category` field in the `classification` document. The returned `category` field remains inside the `classification` document.

```
db.inventory.find(
   { type: 'food', _id: 3 },
   { "classification.category": 1, _id: 0 }
)
```

The operation returns the following document:

```
{ "classification" : { "category" : "chocolate" } }
```

Projection for Array Fields

For fields that contain arrays, MongoDB provides the following projection operators: `$elemMatch`, `$slice`, and `$`.

For example, the `inventory` collection contains the following document:

```
{ "_id" : 5, "type" : "food", "item" : "aaa", "ratings" : [ 5, 8, 9 ] }
```

Then the following operation uses the `$slice` projection operator to return just the first two elements in the `ratings` array.

```
db.inventory.find( { _id: 5 }, { ratings: { $slice: 2 } } )
```

`$elemMatch`, `$slice`, and `$` are the *only* way to project *portions* of an array. For instance, you *cannot* project a portion of an array using the array index; e.g. `{ "ratings.0": 1 }` projection will *not* project the array with the first element.

See also:

Query Documents (page 100)

3.3.6 Limit Number of Elements in an Array after an Update

On this page

New in version 2.4.

Synopsis

Consider an application where users may submit many scores (e.g. for a test), but the application only needs to track the top three test scores.

This pattern uses the `$push` operator with the `$each`, `$sort`, and `$slice` modifiers to sort and maintain an array of fixed size.

Pattern

Consider the following document in the collection `students`:

```
{
  _id: 1,
  scores: [
    { attempt: 1, score: 10 },
    { attempt: 2 , score:8 }
  ]
}
```

The following update uses the `$push` operator with:

- the `$each` modifier to append to the array 2 new elements,
- the `$sort` modifier to order the elements by ascending (`1`) score, and

- the $slice modifier to keep the last 3 elements of the ordered array.

```
db.students.update(
   { _id: 1 },
   {
     $push: {
        scores: {
           $each: [ { attempt: 3, score: 7 }, { attempt: 4, score: 4 } ],
           $sort: { score: 1 },
           $slice: -3
        }
     }
   }
)
```

Note: When using the $sort modifier on the array element, access the field in the embedded document element directly instead of using the *dot notation* on the array field.

After the operation, the document contains only the top 3 scores in the scores array:

```
{
   "_id" : 1,
   "scores" : [
     { "attempt" : 3, "score" : 7 },
     { "attempt" : 2, "score" : 8 },
     { "attempt" : 1, "score" : 10 }
   ]
}
```

See also:

- $push operator,
- $each modifier,
- $sort modifier, and
- $slice modifier.

3.3.7 Iterate a Cursor in the mongo Shell

On this page

The db.collection.find() method returns a cursor. To access the documents, you need to iterate the cursor. However, in the mongo shell, if the returned cursor is not assigned to a variable using the var keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents in the results. The following describes ways to manually iterate the cursor to access the documents or to use the iterator index.

Manually Iterate the Cursor

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

You can call the cursor variable in the shell to iterate up to 20 times [14] and print the matching documents, as in the following example:

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

myCursor
```

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

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

while (myCursor.hasNext()) {
   print(tojson(myCursor.next()));
}
```

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

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

while (myCursor.hasNext()) {
   printjson(myCursor.next());
}
```

You can use the cursor method `forEach()` 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* and your `driver` documentation for more information on cursor methods.

Iterator Index

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

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

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

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

Consider the following example:

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

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

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

[14] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Executing Queries* (page 285) for more information.

3.3.8 Analyze Query Performance

On this page

The `explain()` cursor method provides statistics about the performance of a query. This data output can be useful in measuring if and how a query uses an index. [15]

Evaluate the Performance of a Query

Consider a collection `inventory` with the following documents:

```
{ "_id" : 1, "item" : "f1", type: "food", quantity: 500 }
{ "_id" : 2, "item" : "f2", type: "food", quantity: 100 }
{ "_id" : 3, "item" : "p1", type: "paper", quantity: 200 }
{ "_id" : 4, "item" : "p2", type: "paper", quantity: 150 }
{ "_id" : 5, "item" : "f3", type: "food", quantity: 300 }
{ "_id" : 6, "item" : "t1", type: "toys", quantity: 500 }
{ "_id" : 7, "item" : "a1", type: "apparel", quantity: 250 }
{ "_id" : 8, "item" : "a2", type: "apparel", quantity: 400 }
{ "_id" : 9, "item" : "t2", type: "toys", quantity: 50 }
{ "_id" : 10, "item" : "f4", type: "food", quantity: 75 }
```

Query with No Index

The following query retrieves documents where the `quantity` field has a value between `100` and `200`, inclusive:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } )
```

The query returns the following documents:

```
{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 3, "item" : "p1", "type" : "paper", "quantity" : 200 }
{ "_id" : 4, "item" : "p2", "type" : "paper", "quantity" : 150 }
```

To view the query plan selected, use the `explain()` method:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } ).explain()
```

The `explain()` method returns this output:

```
{
   "cursor" : "BasicCursor",
   "isMultiKey" : false,
   "n" : 3,
   "nscannedObjects" : 10,
   "nscanned" : 10,
   "nscannedObjectsAllPlans" : 10,
   "nscannedAllPlans" : 10,
   "scanAndOrder" : false,
   "indexOnly" : false,
```

[15] Because `explain()` attempts multiple query plans, the `explain()` method does not reflect an accurate timing of query performance. For more information on its behavior, see `explain()`.

```
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "server" : "myMongoDB.local:27017",
    "filterSet" : false
}
```

- `cursor` displays `BasicCursor` to indicate a collection scan.
- `n` displays `3` to indicate that the query matches and returns three documents.
- `nscanned` and `nscannedObjects` display `10` to indicate that MongoDB had to scan ten documents (i.e. all documents in the collection) to find the three matching documents.

The difference between the number of matching documents and the number documents scanned may suggest that, to improve efficiency, the query might benefit from the use of an index.

Query with Index

To support the query on the `quantity` field, add an index on the `quantity` field:

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

To view the query plan statistics, use the `explain()` method:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } ).explain()
```

The `explain()` method returns this output:

```
{
    "cursor" : "BtreeCursor quantity_1",
    "isMultiKey" : false,
    "n" : 3,
    "nscannedObjects" : 3,
    "nscanned" : 3,
    "nscannedObjectsAllPlans" : 3,
    "nscannedAllPlans" : 3,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : { "quantity" : [ [ 100, 200 ] ] },
    "server" : "myMongoDB.local:27017",
    "filterSet" : false
}
```

- `cursor` displays `BtreeCursor quantity_1` to indicate index use and the name of the index.
- `n` displays `3` to indicate that the query matches and returns three documents.
- `nscanned` displays `3` to indicate that MongoDB scanned three index entries.
- `nscannedObjects` displays `3` to indicate that MongoDB scanned three documents.

When run with an index, the query scanned `3` index entries and `3` documents to return `3` matching documents. Without the index, to return the `3` matching documents, the query had to scan the whole collection, scanning `10` documents.

Compare Performance of Indexes

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

Consider the following query:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 300 }, type: "food" } )
```

The query returns the following documents:

```
{ "_id" : 2, "item" : "f2", "type" : "food", "quantity" : 100 }
{ "_id" : 5, "item" : "f3", "type" : "food", "quantity" : 300 }
```

To support the query, add a *compound index* (page 489). With *compound indexes* (page 489), the order of the fields matter.

For example, add the following two compound indexes. The first index orders by `quantity` field first, and then the `type` field. The second index orders by `type` first, and then the `quantity` field.

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

Evaluate the effect of the first index on the query:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 300 }, type: "food" } ).hint({ quantity: 1, type: 1
```

The `explain()` method returns the following output:

```
{
   "cursor" : "BtreeCursor quantity_1_type_1",
   "isMultiKey" : false,
   "n" : 2,
   "nscannedObjects" : 2,
   "nscanned" : 5,
   ...
}
```

MongoDB scanned `5` index keys (`nscanned`) to return `2` matching documents (`n`).

Evaluate the effect of the second index on the query:

```
db.inventory.find( { quantity: { $gte: 100, $lte: 300 }, type: "food" } ).hint({ type: 1, quantity: 1
```

The `explain()` method returns the following output:

```
{
   "cursor" : "BtreeCursor type_1_quantity_1",
   "isMultiKey" : false,
   "n" : 2,
   "nscannedObjects" : 2,
   "nscanned" : 2,
   ...
}
```

MongoDB scanned `2` index keys (`nscanned`) to return `2` matching documents (`n`).

For this example query, the compound index `{ type: 1, quantity: 1 }` is more efficient than the compound index `{ quantity: 1, type: 1 }`.

See also:

Query Optimization (page 70), *Query Plans* (page 72) *Optimize Query Performance* (page 224), *Indexing Strategies* (page 551)

3.3.9 Perform Two Phase Commits

On this page

Synopsis

This document provides a pattern for doing multi-document updates or "multi-document transactions" using a two-phase commit approach for writing data to multiple documents. Additionally, you can extend this process to provide a *rollback-like* (page 124) functionality.

Background

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

Despite the power of single-document atomic operations, there are cases that require multi-document transactions. When executing a transaction composed of sequential operations, certain issues arise, such as:

- 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").
- Consistency: if a major failure (i.e. network, hardware) interrupts the transaction, the database must be able to recover a consistent state.

For situations that require multi-document transactions, you can implement two-phase commit in your application to provide support for these kinds of multi-document updates. Using two-phase commit ensures that data is consistent and, in case of an error, the state that preceded the transaction is *recoverable* (page 124). During the procedure, however, documents can represent pending data and states.

Note: Because only single-document operations are atomic with MongoDB, two-phase commits can only offer transaction-*like* semantics. It is possible for applications to return intermediate data at intermediate points during the two-phase commit or rollback.

Pattern

Overview

Consider a scenario where you want to transfer funds from account `A` to account `B`. In a relational database system, you can subtract the funds from `A` and add the funds to `B` in a single multi-statement transaction. In MongoDB, you

can emulate a two-phase commit to achieve a comparable result.

The examples in this tutorial use the following two collections:

1. A collection named `accounts` to store account information.
2. A collection named `transactions` to store information on the fund transfer transactions.

Initialize Source and Destination Accounts

Insert into the `accounts` collection a document for account `A` and a document for account `B`.

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

The operation returns a `BulkWriteResult()` object with the status of the operation. Upon successful insert, the `BulkWriteResult()` has `nInserted` set to `2` .

Initialize Transfer Record

For each fund transfer to perform, insert into the `transactions` collection a document with the transfer information. The document contains the following fields:

- `source` and `destination` fields, which refer to the `_id` fields from the `accounts` collection,
- `value` field, which specifies the amount of transfer affecting the `balance` of the `source` and `destination` accounts,
- `state` field, which reflects the current state of the transfer. The `state` field can have the value of `initial`, `pending`, `applied`, `done`, `canceling`, and `canceled`.
- `lastModified` field, which reflects last modification date.

To initialize the transfer of `100` from account `A` to account `B`, insert into the `transactions` collection a document with the transfer information, the transaction `state` of `"initial"`, and the `lastModified` field set to the current date:

```
db.transactions.insert(
    { _id: 1, source: "A", destination: "B", value: 100, state: "initial", lastModified: new Date() }
)
```

The operation returns a `WriteResult()` object with the status of the operation. Upon successful insert, the `WriteResult()` object has `nInserted` set to `1`.

Transfer Funds Between Accounts Using Two-Phase Commit

Step 1: Retrieve the transaction to start. From the `transactions` collection, find a transaction in the `initial` state. Currently the `transactions` collection has only one document, namely the one added in the *Initialize Transfer Record* (page 121) step. If the collection contains additional documents, the query will return any transaction with an `initial` state unless you specify additional query conditions.

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

Type the variable `t` in the `mongo` shell to print the contents of the variable. The operation should print a document similar to the following except the `lastModified` field should reflect date of your insert operation:

```
{ "_id" : 1, "source" : "A", "destination" : "B", "value" : 100, "state" : "initial", "lastModified"
```

Step 2: Update transaction state to pending. Set the transaction `state` from `initial` to `pending` and use the `$currentDate` operator to set the `lastModified` field to the current date.

```
db.transactions.update(
    { _id: t._id, state: "initial" },
    {
      $set: { state: "pending" },
      $currentDate: { lastModified: true }
    }
)
```

The operation returns a `WriteResult()` object with the status of the operation. Upon successful update, the `nMatched` and `nModified` displays `1`.

In the update statement, the `state: "initial"` condition ensures that no other process has already updated this record. If `nMatched` and `nModified` is `0`, go back to the first step to get a different transaction and restart the procedure.

Step 3: Apply the transaction to both accounts. Apply the transaction `t` to both accounts using the `update()` method *if* the transaction has not been applied to the accounts. In the update condition, include the condition `pendingTransactions: { $ne: t._id }` in order to avoid re-applying the transaction if the step is run more than once.

To apply the transaction to the account, update both the `balance` field and the `pendingTransactions` field.

Update the source account, subtracting from its `balance` the transaction `value` and adding to its `pendingTransactions` array the transaction `_id`.

```
db.accounts.update(
   { _id: t.source, pendingTransactions: { $ne: t._id } },
   { $inc: { balance: -t.value }, $push: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Update the destination account, adding to its `balance` the transaction `value` and adding to its `pendingTransactions` array the transaction `_id`.

```
db.accounts.update(
   { _id: t.destination, pendingTransactions: { $ne: t._id } },
   { $inc: { balance: t.value }, $push: { pendingTransactions: t._id } }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 4: Update transaction state to applied. Use the following `update()` operation to set the transaction's `state` to `applied` and update the `lastModified` field:

```
db.transactions.update(
   { _id: t._id, state: "pending" },
   {
     $set: { state: "applied" },
     $currentDate: { lastModified: true }
```

```
    }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 5: Update both accounts' list of pending transactions. Remove the applied transaction `_id` from the `pendingTransactions` array for both accounts.

Update the source account.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Update the destination account.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 6: Update transaction state to done. Complete the transaction by setting the `state` of the transaction to `done` and updating the `lastModified` field:

```
db.transactions.update(
   { _id: t._id, state: "applied" },
   {
     $set: { state: "done" },
     $currentDate: { lastModified: true }
   }
)
```

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

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 successfully. This section presents an overview of possible failures and provides steps to recover from these kinds of events.

Recovery Operations

The two-phase commit pattern allows applications running the sequence to resume the transaction and arrive at a consistent state. Run the recovery operations at application startup, and possibly at regular intervals, to catch any unfinished transactions.

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

The following recovery procedures uses the `lastModified` date as an indicator of whether the pending transaction requires recovery; specifically, if the pending or applied transaction has not been updated in the last 30 minutes,

the procedures determine that these transactions require recovery. You can use different conditions to make this determination.

Transactions in Pending State To recover from failures that occur after step "Update transaction state to pending. (page **??**)" but before "Update transaction state to applied. (page **??**)" step, retrieve from the `transactions` collection a pending transaction for recovery:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

var t = db.transactions.findOne( { state: "pending", lastModified: { $lt: dateThreshold } } );
```

And resume from step "Apply the transaction to both accounts. (page **??**)"

Transactions in Applied State To recover from failures that occur after step "Update transaction state to applied. (page **??**)" but before "Update transaction state to done. (page **??**)" step, retrieve from the `transactions` collection an applied transaction for recovery:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

var t = db.transactions.findOne( { state: "applied", lastModified: { $lt: dateThreshold } } );
```

And resume from "Update both accounts' list of pending transactions. (page **??**)"

Rollback Operations

In some cases, you may need to "roll back" or undo a transaction; e.g., if the application needs to "cancel" the transaction or if one of the accounts does not exist or stops existing during the transaction.

Transactions in Applied State After the "Update transaction state to applied. (page **??**)" step, you should **not** roll back the transaction. Instead, complete that transaction and *create a new transaction* (page 121) to reverse the transaction by switching the values in the source and the destination fields.

Transactions in Pending State After the "Update transaction state to pending. (page **??**)" step, but before the "Update transaction state to applied. (page **??**)" step, you can rollback the transaction using the following procedure:

Step 1: Update transaction state to canceling. Update the transaction `state` from `pending` to `canceling`.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Step 2: Undo the transaction on both accounts. To undo the transaction on both accounts, reverse the transaction `t` if the transaction has been applied. In the update condition, include the condition `pendingTransactions: t._id` in order to update the account only if the pending transaction has been applied.

Update the destination account, subtracting from its `balance` the transaction `value` and removing the transaction `_id` from the `pendingTransactions` array.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`. If the pending transaction has not been previously applied to this account, no document will match the update condition and `nMatched` and `nModified` will be `0`.

Update the source account, adding to its `balance` the transaction `value` and removing the transaction `_id` from the `pendingTransactions` array.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`. If the pending transaction has not been previously applied to this account, no document will match the update condition and `nMatched` and `nModified` will be `0`.

Step 3: Update transaction state to canceled. To finish the rollback, update the transaction `state` from `canceling` to `cancelled`.

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

Upon successful update, the method returns a `WriteResult()` object with `nMatched` and `nModified` set to `1`.

Multiple Applications

Transactions exist, in part, so that multiple applications can create and run operations concurrently without causing data inconsistency or conflicts. In our procedure, to update or retrieve the transaction document, the update conditions include a condition on the `state` field to prevent reapplication of the transaction by multiple applications.

For example, applications `App1` and `App2` both grab the same transaction, which is in the `initial` state. `App1` applies the whole transaction before `App2` starts. When `App2` attempts to perform the "Update transaction state to pending. (page **??**)" step, the update condition, which includes the `state: "initial"` criterion, will not match any document, and the `nMatched` and `nModified` will be `0`. This should signal to `App2` to go back to the first step to restart the procedure with a different transaction.

When multiple applications are running, it is crucial that only one application can handle a given transaction at any point in time. As such, in addition including the expected state of the transaction in the update condition, you can also create a marker in the transaction document itself to identify the application that is handling the transaction. Use `findAndModify()` method to modify the transaction and get it back in one step:

```
t = db.transactions.findAndModify(
       {
         query: { state: "initial", application: { $exists: false } },
         update:
           {
             $set: { state: "pending", application: "App1" },
             $currentDate: { lastModified: true }
           },
         new: true
       }
    )
```

Amend the transaction operations to ensure that only applications that match the identifier in the `application` field apply the transaction.

If the application `App1` fails during transaction execution, you can use the *recovery procedures* (page 123), but applications should ensure that they "own" the transaction before applying the transaction. For example to find and resume the pending job, use a query that resembles the following:

```
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);

db.transactions.find(
   {
     application: "App1",
     state: "pending",
     lastModified: { $lt: dateThreshold }
   }
)
```

Using Two-Phase Commits in Production Applications

The example transaction above is intentionally simple. For example, it assumes that it is always possible to roll back operations to an account and that account balances can hold negative values.

Production implementations would likely be more complex. Typically, accounts need information about current balance, pending credits, and pending debits.

For all transactions, ensure that you use the appropriate level of *write concern* (page 82) for your deployment.

3.3.10 Update Document if Current

On this page

Overview

The *Update if Current* pattern is an approach to *concurrency control* (page 86) when multiple applications have access to the data.

Pattern

The pattern queries for the document to update. Then, for each field to modify, the pattern includes the field and its value in the returned document in the query predicate for the update operation. This way, the update only modifies the document fields *if* the fields have not changed since the query.

Example

Consider the following example in the `mongo` shell. The example updates the `quantity` and the `reordered` fields of a document *only* if the fields have not changed since the query.

Changed in version 2.6: The `db.collection.update()` method now returns a `WriteResult()` object that contains the status of the operation. Previous versions required an extra `db.getLastErrorObj()` method call.

```
var myDocument = db.products.findOne( { sku: "abc123" } );

if ( myDocument ) {
   var oldQuantity = myDocument.quantity;
   var oldReordered = myDocument.reordered;

   var results = db.products.update(
      {
        _id: myDocument._id,
        quantity: oldQuantity,
        reordered: oldReordered
      },
      {
        $inc: { quantity: 50 },
        $set: { reordered: true }
      }
   )

   if ( results.hasWriteError() ) {
      print( "unexpected error updating document: " + tojson(results) );
   }
   else if ( results.nMatched === 0 ) {
      print( "No matching document for " +
             "{ _id: "+ myDocument._id.toString() +
             ", quantity: " + oldQuantity +
             ", reordered: " + oldReordered
             + " } "
      );
   }
}
```

Modifications to the Pattern

Another approach is to add a `version` field to the documents. Applications increment this field upon each update operation to the documents. You must be able to ensure that *all* clients that connect to your database include the

version field in the query predicate. To associate increasing numbers with documents in a collection, you can use one of the methods described in *Create an Auto-Incrementing Sequence Field* (page 130).

For more approaches, see *Concurrency Control* (page 86).

3.3.11 Create Tailable Cursor

On this page

Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for *capped collections* (page 219) 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 that have high write volumes where indexes aren't practical. For instance, MongoDB *replication* (page 563) 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 that document.

  A *dead* cursor has an id of `0`.

See your `driver documentation` for the driver-specific method to specify the tailable cursor.

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()` to ensure that the query uses the `$natural` order.
- In an outer `while(1)` loop,
  - Query the capped collection and return a tailable cursor that blocks for several seconds waiting for new documents

    ```
    auto_ptr<DBClientCursor> c =
         conn.query(ns, query, 0, 0, 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[16]

3.3.12 Create an Auto-Incrementing Sequence Field

On this page

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 506). 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:

- *Use Counters Collection* (page 131)
- *Optimistic Loop* (page 132)

Considerations

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`.

[16] http://shtylman.com/post/the-tail-of-mongodb

Procedures

Use Counters Collection

Counter Collection Implementation Use a separate `counters` collection to track the *last* number sequence used. The `_id` field contains the sequence name and the `seq` field contains the last value of the sequence.

1. Insert into the `counters` collection, the initial value for the `userid`:

```
db.counters.insert(
   {
      _id: "userid",
      seq: 0
   }
)
```

2. Create a `getNextSequence` function that accepts a `name` of the sequence. The function uses the `findAndModify()` method to atomically increment the `seq` value and return this new value:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true
          }
   );

   return ret.seq;
}
```

3. Use this `getNextSequence()` function during `insert()`.

```
db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Sarah C."
   }
)

db.users.insert(
   {
     _id: getNextSequence("userid"),
     name: "Bob D."
   }
)
```

You can verify the results with `find()`:

```
db.users.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id : 1,
  name : "Sarah C."
}
{
  _id : 2,
```

```
    name : "Bob D."
}
```

findAndModify Behavior When `findAndModify()` includes the `upsert: true` option **and** the query field(s) is not uniquely indexed, the method could insert a document multiple times in certain circumstances. For instance, if multiple clients each invoke the method with the same query condition and these methods complete the find phase before any of methods perform the modify phase, these methods could insert the same document.

In the `counters` collection example, the query field is the `_id` field, which always has a unique index. Consider that the `findAndModify()` includes the `upsert: true` option, as in the following modified example:

```
function getNextSequence(name) {
   var ret = db.counters.findAndModify(
          {
            query: { _id: name },
            update: { $inc: { seq: 1 } },
            new: true,
            upsert: true
          }
   );

   return ret.seq;
}
```

If multiple clients were to invoke the `getNextSequence()` method with the same `name` parameter, then the methods would observe one of the following behaviors:

- Exactly one `findAndModify()` would successfully insert a new document.
- Zero or more `findAndModify()` methods would update the newly inserted document.
- Zero or more `findAndModify()` methods would fail when they attempted to insert a duplicate.

If the method fails due to a unique index constraint violation, retry the method. Absent a delete of the document, the retry should not fail.

Optimistic Loop

In this pattern, an *Optimistic Loop* calculates the incremented `_id` value and attempts to insert a document with the calculated `_id` value. If the insert is successful, the loop ends. Otherwise, the loop will iterate through possible `_id` values until the insert is successful.

1. Create a function named `insertDocument` that performs the "insert if not present" loop. The function wraps the `insert()` method and takes a `doc` and a `targetCollection` arguments.

   Changed in version 2.6: The `db.collection.insert()` method now returns a *writeresults-insert* object that contains the status of the operation. Previous versions required an extra `db.getLastErrorObj()` method call.

   ```
   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;
   ```

```
        var results = targetCollection.insert(doc);

        if( results.hasWriteError() ) {
            if( results.writeError.code == 11000 /* dup key */ )
                continue;
            else
                print( "unexpected error inserting data: " + tojson( results ) );
        }

        break;
    }
}
```

The `while (1)` loop performs the following actions:

- Queries the `targetCollection` for the document with the maximum `_id` value.
- Determines the next sequence value for `_id` by:
  - adding `1` to the returned `_id` value if the returned cursor points to a document.
  - otherwise: it sets the next sequence value to `1` if the returned cursor points to no document.
- For the `doc` to insert, set its `_id` field to the calculated sequence value `seq`.
- Insert the `doc` into the `targetCollection`.
- If the insert operation errors with duplicate key, repeat the loop. Otherwise, if the insert operation encounters some other error or if the operation succeeds, break out of the loop.

2. Use the `insertDocument()` function to perform an insert:

```
var myCollection = db.users2;

insertDocument(
   {
     name: "Grace H."
   },
   myCollection
);

insertDocument(
   {
     name: "Ted R."
   },
   myCollection
)
```

You can verify the results with `find()`:

```
db.users2.find()
```

The `_id` fields contain incrementing sequence values:

```
{
  _id: 1,
  name: "Grace H."
}
{
  _id : 2,
  "name" : "Ted R."
}
```

The `while` loop may iterate many times in collections with larger insert volumes.

3.4 MongoDB CRUD Reference

On this page

3.4.1 Query Cursor Methods

Name	Description
`cursor.count()`	Returns a count of the documents in a cursor.
`cursor.explain()`	Reports on the query execution plan, including index use, for a cursor.
`cursor.hint()`	Forces MongoDB to use a specific index for a query.
`cursor.limit()`	Constrains the size of a cursor's result set.
`cursor.next()`	Returns the next document in a cursor.
`cursor.skip()`	Returns a cursor that begins returning results only after passing or skipping a number of documents.
`cursor.sort()`	Returns results ordered according to a sort specification.
`cursor.toArray()`	Returns an array that contains all documents returned by the cursor.

3.4.2 Query and Data Manipulation Collection Methods

Name	Description
`db.collection.count()`	Wraps `count` to return a count of the number of documents in a collection or matching a query.
`db.collection.distinct()`	Returns an array of documents that have distinct values for the specified field.
`db.collection.find()`	Performs a query on a collection and returns a cursor object.
`db.collection.findOne()`	Performs a query and returns a single document.
`db.collection.insert()`	Creates a new document in a collection.
`db.collection.remove()`	Deletes documents from a collection.
`db.collection.save()`	Provides a wrapper around an `insert()` and `update()` to insert new documents.
`db.collection.update()`	Modifies a document in a collection.

3.4.3 MongoDB CRUD Reference Documentation

Write Concern Reference **(page 135)** Configuration options associated with the guarantee MongoDB provides when reporting on the success of a write operation.

SQL to MongoDB Mapping Chart **(page 136)** An overview of common database operations showing both the MongoDB operations and SQL statements.

The bios Example Collection **(page 142)** Sample data for experimenting with MongoDB. `insert()`, `update()` and `find()` pages use the data for some of their examples.

Write Concern Reference

On this page

Write concern (page 82) describes the guarantee that MongoDB provides when reporting on the success of a write operation.

Changed in version 2.6: A new protocol for *write operations* (page 832) integrates write concerns with the write operations and eliminates the need to call the `getLastError` command. Previous versions required a `getLastError` command immediately after a write operation to specify the write concern.

Available Write Concern

Write concern can include the *w* (page 135) option to specify the required number of acknowledgments before returning, the *j* (page 136) option to require writes to the journal before returning, and *wtimeout* (page 136) option to specify a time limit to prevent write operations from blocking indefinitely.

In sharded clusters, `mongos` instances will pass the write concern on to the shard.

w Option The `w` option provides the ability to disable write concern entirely *as well as* specify the write concern for *replica sets*.

MongoDB uses `w: 1` as the default write concern. `w: 1` provides basic receipt acknowledgment.

The `w` option accepts the following values:

Value	Description
`1`	Provides acknowledgment of write operations on a standalone `mongod` or the *primary* in a replica set. This is the default write concern for MongoDB.
`0`	Disables basic acknowledgment of write operations, but returns information about socket exceptions and networking errors to the application. If you disable basic write operation acknowledgment but require journal commit acknowledgment, the journal commit prevails, and the server will require that `mongod` acknowledge the write operation.
<Number greater than 1>	Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. For example, `w: 2` indicates acknowledgements from the primary and at least one secondary. 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 allows you to avoid hard coding assumptions about the size of your replica set into your application. Changed in version 2.6: In *Master/Slave* (page 600) deployments, MongoDB treats `w: "majority"` as equivalent to `w: 1`. In earlier versions of MongoDB, `w: "majority"` produces an error in *master/slave* (page 600) deployments.
<tag set>	By specifying a *tag set* (page 641), you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.

j Option The `j` option confirms that the `mongod` instance has written the data to the on-disk journal. This ensures that data is not lost if the `mongod` instance shuts down unexpectedly. Set to `true` to enable.

Changed in version 2.6: Specifying a write concern that includes `j: true` to a `mongod` or `mongos` running with `--nojournal` option now errors. Previous versions would ignore the `j: true`.

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.

---

wtimeout This option specifies a time limit, in milliseconds, for the write concern. `wtimeout` is only applicable for `w` values greater than `1`.

`wtimeout` causes write operations to return with an error after the specified limit, even if the required write concern will eventually succeed. When these write operations return, MongoDB **does not** undo successful data modifications performed before the write concern exceeded the `wtimeout` time limit.

If you do not specify the `wtimeout` option and the level of write concern is unachievable, the write operation will block indefinitely. Specifying a `wtimeout` value of `0` is equivalent to a write concern without the `wtimeout` option.

See also:

Write Concern Introduction (page 82) and *Write Concern for Replica Sets* (page 83).

SQL to MongoDB Mapping Chart

On this page

In addition to the charts that follow, you might want to consider the *Frequently Asked Questions* (page 761) section for a selection of common questions about MongoDB.

Terminology and Concepts

The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts.

SQL Terms/Concepts	MongoDB Terms/Concepts
database	*database*
table	*collection*
row	*document* or *BSON* document
column	*field*
index	*index*
table joins	embedded documents and linking
primary key	*primary key*
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the *_id* field.
aggregation (e.g. group by)	aggregation pipeline See the *SQL to Aggregation Mapping Chart* (page 477).

Executables

The following table presents some database executables and the corresponding MongoDB executables. This table is *not* meant to be exhaustive.

	MongoDB	MySQL	Oracle	Informix	DB2
Database Server	`mongod`	`mysqld`	`oracle`	`IDS`	`DB2 Server`
Database Client	`mongo`	`mysql`	`sqlplus`	`DB-Access`	`DB2 Client`

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

`CREATE TABLE users (`
`  id MEDIUMINT NOT NULL`
`  AUTO_INCREMENT,`
`  user_id Varchar(30),`
`  age Number,`
`  status char(1),`
`  PRIMARY KEY (id)`
`)`
	Implicitly created on first `insert()` operation. The primary key `_id` is automatically added if `_id` field is not specified.
	`db.users.insert( {`
	`  user_id: "abc123",`
	`  age: 55,`
	`  status: "A"`
	`} )`
	However, you can also explicitly create a collection:
	`db.createCollection("users")`

`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()` operations can add fields to existing documents using the `$set` operator.
	`db.users.update(`
	`  { },`
	`  { $set: { join_date: new Date() } },`
	`  { multi: true }`
	`)`

`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()` operations can remove fields from documents using the `$unset` operator.
	`db.users.update(`
	`  { },`
	`  { $unset: { join_date: "" } },`
	`  { multi: true }`
	`)`

`CREATE INDEX idx_user_id_asc` `ON users(user_id)`	`db.users.ensureIndex( { user_id: 1 } )`

`CREATE INDEX`
`  idx_user_id_asc_age_desc`
`ON users(user_id, age DESC)`
	`db.users.ensureIndex( { user_id: 1, age: -1 } )`

`DROP TABLE users`	`db.users.drop()`

For more information, see `db.collection.insert()`, `db.createCollection()`, `db.collection.update()`, `$set`, `$unset`, `db.collection.ensureIndex()`, *indexes* (page 485), `db.collection.drop()`, and *Data Modeling Concepts* (page 151).

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

`INSERT INTO users(user_id,`
`  age,`
`  status)`
`VALUES ("bcd001",`
`  45,`
`  "A")`
	`db.users.insert(`
	`  { user_id: "bcd001", age: 45, status: "A" }`
	`)`

For more information, see `db.collection.insert()`.

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
`SELECT * FROM users`	`db.users.find()`
`SELECT id, user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1 } )`
`SELECT user_id, status FROM users`	`db.users.find( { }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status = "A"`	`db.users.find( { status: "A" } )`
`SELECT user_id, status FROM users WHERE status = "A"`	`db.users.find( { status: "A" }, { user_id: 1, status: 1, _id: 0 } )`
`SELECT * FROM users WHERE status != "A"`	`db.users.find( { status: { $ne: "A" } } )`
`SELECT * FROM users WHERE status = "A" AND age = 50`	`db.users.find( { status: "A", age: 50 } )`
`SELECT * FROM users WHERE status = "A" OR age = 50`	`db.users.find( { $or: [ { status: "A" } , { age: 50 } ] } )`
`SELECT * FROM users WHERE age > 25`	`db.users.find( { age: { $gt: 25 } } )`
`SELECT * FROM users WHERE age < 25`	`db.users.find( { age: { $lt: 25 } } )`
`SELECT * FROM users WHERE age > 25 AND age <= 50`	`db.users.find( { age: { $gt: 25, $lte: 50 } } )`
`SELECT * FROM users WHERE user_id like "%bc%"`	`db.users.find( { user_id: /bc/ } )`

For more information, see `db.collection.find()`, `db.collection.distinct()`, `db.collection.findOne()`, `$ne` `$and`, `$or`, `$gt`, `$lt`, `$exists`, `$lte`, `$regex`, `limit()`, `skip()`, `explain()`, `sort()`, and `count()`.

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

`UPDATE users`
`SET status = "C"`
`WHERE age > 25`
	`db.users.update(`
	`{ age: { $gt: 25 } },`
	`{ $set: { status: "C" } },`
	`{ multi: true }`
	`)`

`UPDATE users`
`SET age = age + 3`
`WHERE status = "A"`
	`db.users.update(`
	`{ status: "A" } ,`
	`{ $inc: { age: 3 } },`
	`{ multi: true }`
	`)`

For more information, see `db.collection.update()`, `$set`, `$inc`, and `$gt`.

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
`DELETE FROM users` `WHERE status = "D"`	`db.users.remove( { status: "D" } )`
`DELETE FROM users`	`db.users.remove({})`

For more information, see `db.collection.remove()`.

Additional Resources

- Transitioning from SQL to MongoDB (Presentation)[17]
- Best Practices for Migrating from RDBMS to MongoDB (Webinar)[18]
- RDBMS to MongoDB Migration Guide[19]
- SQL vs. MongoDB Day 1-2[20]
- SQL vs. MongoDB Day 3-5[21]
- MongoDB vs. SQL Day 18[22]

[17] http://www.mongodb.com/presentations/webinar-transitioning-sql-mongodb?jmp=docs
[18] http://www.mongodb.com/webinar/best-practices-migration?jmp=docs
[19] http://www.mongodb.com/lp/white-paper/migration-rdbms-nosql-mongodb?jmp=docs
[20] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-1-2?jmp=docs
[21] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5?jmp=docs
[22] http://www.mongodb.com/blog/post/mongodb-vs-sql-day-14?jmp=docs

- MongoDB and MySQL Compared[23]
- MongoDB Database Modernization Consulting Package[24]

The `bios` Example Collection

The `bios` collection provides example data for experimenting with MongoDB. Many of this guide's examples on `insert`, `update` and `read` operations create or query data from the `bios` collection.

The following documents comprise the `bios` collection. In the examples, the data might be different, as the examples themselves make changes to the data.

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

{
    "_id" : ObjectId("51df07b094c6acd67e492f41"),
    "name" : {
        "first" : "John",
        "last" : "McCarthy"
    },
```

[23] http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
[24] https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

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

{
    "_id" : 3,
    "name" : {
        "first" : "Grace",
        "last" : "Hopper"
    },
    "title" : "Rear Admiral",
    "birth" : ISODate("1906-12-09T05:00:00Z"),
    "death" : ISODate("1992-01-01T05:00:00Z"),
    "contribs" : [
        "UNIVAC",
        "compiler",
        "FLOW-MATIC",
        "COBOL"
    ],
    "awards" : [
        {
            "award" : "Computer Sciences Man of the Year",
            "year" : 1969,
            "by" : "Data Processing Management Association"
        },
        {
            "award" : "Distinguished Fellow",
            "year" : 1973,
            "by" : " British Computer Society"
        },
        {
            "award" : "W. W. McDowell Award",
            "year" : 1976,
            "by" : "IEEE Computer Society"
        },
        {
```

```
            "award" : "National Medal of Technology",
            "year" : 1991,
            "by" : "United States"
        }
    ]
}

{
    "_id" : 4,
    "name" : {
        "first" : "Kristen",
        "last" : "Nygaard"
    },
    "birth" : ISODate("1926-08-27T04:00:00Z"),
    "death" : ISODate("2002-08-10T04:00:00Z"),
    "contribs" : [
        "OOP",
        "Simula"
    ],
    "awards" : [
        {
            "award" : "Rosing Prize",
            "year" : 1999,
            "by" : "Norwegian Data Association"
        },
        {
            "award" : "Turing Award",
            "year" : 2001,
            "by" : "ACM"
        },
        {
            "award" : "IEEE John von Neumann Medal",
            "year" : 2001,
            "by" : "IEEE"
        }
    ]
}

{
    "_id" : 5,
    "name" : {
        "first" : "Ole-Johan",
        "last" : "Dahl"
    },
    "birth" : ISODate("1931-10-12T04:00:00Z"),
    "death" : ISODate("2002-06-29T04:00:00Z"),
    "contribs" : [
        "OOP",
        "Simula"
    ],
    "awards" : [
        {
            "award" : "Rosing Prize",
            "year" : 1999,
            "by" : "Norwegian Data Association"
        },
        {
            "award" : "Turing Award",
```

```
            "year" : 2001,
            "by" : "ACM"
        },
        {
            "award" : "IEEE John von Neumann Medal",
            "year" : 2001,
            "by" : "IEEE"
        }
    ]
}

{
    "_id" : 6,
    "name" : {
        "first" : "Guido",
        "last" : "van Rossum"
    },
    "birth" : ISODate("1956-01-31T05:00:00Z"),
    "contribs" : [
        "Python"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : 2001,
            "by" : "Free Software Foundation"
        },
        {
            "award" : "NLUUG Award",
            "year" : 2003,
            "by" : "NLUUG"
        }
    ]
}

{
    "_id" : ObjectId("51e062189c6ae665454e301d"),
    "name" : {
        "first" : "Dennis",
        "last" : "Ritchie"
    },
    "birth" : ISODate("1941-09-09T04:00:00Z"),
    "death" : ISODate("2011-10-12T04:00:00Z"),
    "contribs" : [
        "UNIX",
        "C"
    ],
    "awards" : [
        {
            "award" : "Turing Award",
            "year" : 1983,
            "by" : "ACM"
        },
        {
            "award" : "National Medal of Technology",
            "year" : 1998,
            "by" : "United States"
        },
```

```
        {
            "award" : "Japan Prize",
            "year" : 2011,
            "by" : "The Japan Prize Foundation"
        }
    ]
}

{
    "_id" : 8,
    "name" : {
        "first" : "Yukihiro",
        "aka" : "Matz",
        "last" : "Matsumoto"
    },
    "birth" : ISODate("1965-04-14T04:00:00Z"),
    "contribs" : [
        "Ruby"
    ],
    "awards" : [
        {
            "award" : "Award for the Advancement of Free Software",
            "year" : "2011",
            "by" : "Free Software Foundation"
        }
    ]
}

{
    "_id" : 9,
    "name" : {
        "first" : "James",
        "last" : "Gosling"
    },
    "birth" : ISODate("1955-05-19T04:00:00Z"),
    "contribs" : [
        "Java"
    ],
    "awards" : [
        {
            "award" : "The Economist Innovation Award",
            "year" : 2002,
            "by" : "The Economist"
        },
        {
            "award" : "Officer of the Order of Canada",
            "year" : 2007,
            "by" : "Canada"
        }
    ]
}

{
    "_id" : 10,
    "name" : {
        "first" : "Martin",
        "last" : "Odersky"
    },
```

```
    "contribs" : [
        "Scala"
    ]
}
```

CHAPTER 4

Data Models

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. This flexibility gives you data-modeling choices to match your application and its performance requirements.

Data Modeling Introduction **(page 149)** An introduction to data modeling in MongoDB.

Data Modeling Concepts **(page 151)** The core documentation detailing the decisions you must make when determining a data model, and discussing considerations that should be taken into account.

Data Model Examples and Patterns **(page 158)** Examples of possible data models that you can use to structure your MongoDB documents.

Data Model Reference **(page 176)** Reference material for data modeling for developers of MongoDB applications.

4.1 Data Modeling Introduction

On this page

Data in MongoDB has a *flexible schema*. Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's *collections* do not enforce *document* structure. This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.

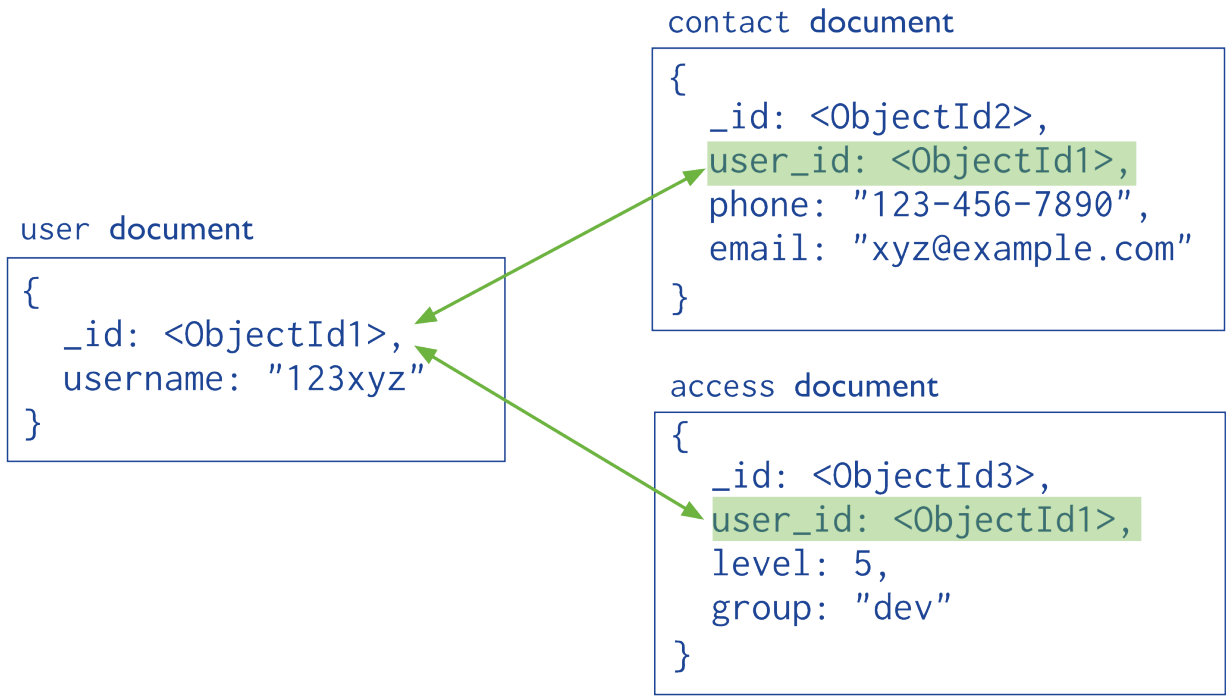
The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

4.1.1 Document Structure

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. There are two tools that allow applications to represent these relationships: *references* and *embedded documents*.

References

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these *references* (page 179) to access the related data. Broadly, these are *normalized* data models.



See *Normalized Data Models* (page 153) for the strengths and weaknesses of using references.

Embedded Data

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

See *Embedded Data Models* (page 152) for the strengths and weaknesses of embedding documents.

4.1.2 Atomicity of Write Operations

In MongoDB, write operations are atomic at the *document* level, and no single write operation can atomically affect more than one document or more than one collection. A denormalized data model with embedded data combines all related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.

However, schemas that facilitate atomic writes may limit ways that applications can use the data or may limit ways to modify applications. The *Atomicity Considerations* (page 154) documentation describes the challenge of designing a schema that balances flexibility and atomicity.

4.1.3 Document Growth

Some updates, such as pushing elements to an array or adding new fields, increase a *document's* size. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. The growth consideration can affect the decision to normalize or denormalize data. See *Document Growth Considerations* (page 154) for more about planning for and managing document growth in MongoDB.

4.1.4 Data Use and Performance

When designing a data model, consider how applications will use your database. For instance, if your application only uses recently inserted documents, consider using *Capped Collections* (page 219). Or if your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance.

See *Operational Factors and Data Models* (page 154) for more information on these and other operational considerations that affect data model designs.

4.2 Data Modeling Concepts

Consider the following aspects of data modeling in MongoDB:

Data Model Design **(page 152)** Presents the different strategies that you can choose from when determining your data model, their strengths and their weaknesses.

Operational Factors and Data Models **(page 154)** Details features you should keep in mind when designing your data model, such as lifecycle management, indexing, horizontal scalability, and document growth.

GridFS **(page 156)** GridFS is a specification for storing documents that exceeds the *BSON*-document size limit of 16MB.

For a general introduction to data modeling in MongoDB, see the *Data Modeling Introduction* (page 149). For example data models, see *Data Modeling Examples and Patterns* (page 158).

4.2.1 Data Model Design

On this page

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to *embed* (page 152) or to *use references* (page 153).

Embedded Data Models

With MongoDB, you may embed related data in a single structure or document. These schema are generally known as "denormalized" models, and take advantage of MongoDB's rich documents. Consider the following diagram:

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, use embedded data models when:

- you have "contains" relationships between entities. See *Model One-to-One Relationships with Embedded Documents* (page 159).
- you have one-to-many relationships between entities. In these relationships the "many" or child documents always appear with or are viewed in the context of the "one" or parent documents. See *Model One-to-Many Relationships with Embedded Documents* (page 160).

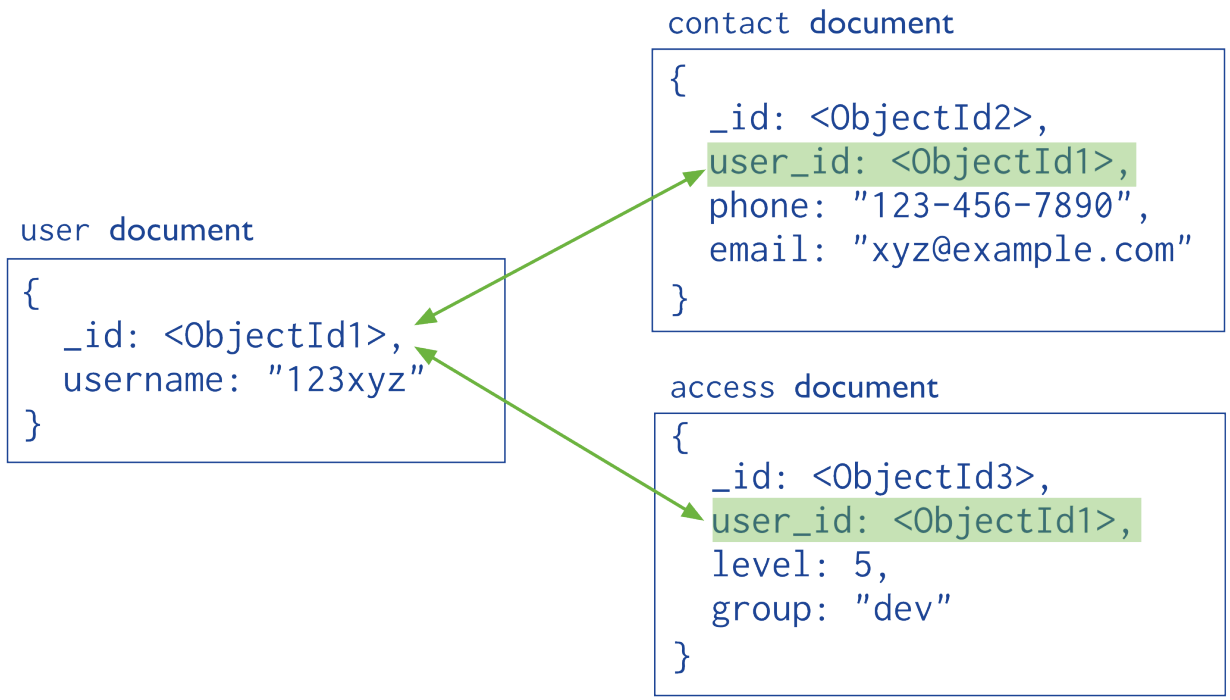
In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.

However, embedding related data in documents may lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. See *Document Growth* (page 154) for details. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size`. For bulk binary data, consider *GridFS* (page 156).

To interact with embedded documents, use *dot notation* to "reach into" embedded documents. See *query for data in arrays* (page 103) and *query data in embedded documents* (page 102) for more examples on accessing data in arrays and embedded documents.

Normalized Data Models

Normalized data models describe relationships using *references* (page 179) between documents.



In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

References provides more flexibility than embedding. However, client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more round trips to the server.

See *Model One-to-Many Relationships with Document References* (page 161) for an example of referencing. For examples of various tree models using references, see *Model Tree Structures* (page 163).

Additional Resources

- Thinking in Documents (Presentation)[1]
- Schema Design for Time Series Data (Presentation)[2]
- Socialite, the Open Source Status Feed - Storing a Social Graph (Presentation)[3]
- MongoDB Rapid Start Consultation Services[4]

4.2.2 Operational Factors and Data Models

On this page

Modeling application data for MongoDB depends on both the data itself, as well as the characteristics of MongoDB itself. For example, different data models may allow applications to use more efficient queries, increase the throughput of insert and update operations, or distribute activity to a sharded cluster more effectively.

These factors are *operational* or address requirements that arise outside of the application but impact the performance of MongoDB based applications. When developing a data model, analyze all of your application's *read operations* (page 64) and *write operations* (page 77) in conjunction with the following considerations.

Document Growth

Some updates to documents can increase the size of documents. These updates include pushing elements to an array (i.e. `$push`) and adding new fields to a document. If the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. Relocating documents takes longer than *in place updates* and can lead to fragmented storage. Although MongoDB automatically *adds padding to document allocations* (page 95) to minimize the likelihood of relocation, data models should avoid document growth when possible.

For instance, if your applications require updates that will cause document growth, you may want to refactor your data model to use references between data in distinct documents rather than a denormalized data model.

MongoDB adaptively adjusts the amount of automatic padding to reduce occurrences of relocation. You may also use a *pre-allocation* strategy to explicitly avoid document growth. Refer to the Pre-Aggregated Reports Use Case[5] for an example of the *pre-allocation* approach to handling document growth.

See *Storage* (page 94) for more information on MongoDB's storage model and record allocation strategies.

Atomicity

In MongoDB, operations are atomic at the *document* level. No **single** write operation can change more than one document. Operations that modify more than a single document in a collection still operate on one document at a time.

[1] http://www.mongodb.com/presentations/webinar-back-basics-1-thinking-documents?jmp=docs
[2] http://www.mongodb.com/presentations/webinar-time-series-data-mongodb?jmp=docs
[3] http://www.mongodb.com/presentations/socialite-open-source-status-feed-part-2-managing-social-graph?jmp=docs
[4] https://www.mongodb.com/products/consulting?jmp=docs#rapid_start
[5] https://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports

[6] Ensure that your application stores all fields with atomic dependency requirements in the same document. If the application can tolerate non-atomic updates for two pieces of data, you can store these data in separate documents.

A data model that embeds related data in a single document facilitates these kinds of atomic operations. For data models that store references between related pieces of data, the application must issue separate read and write operations to retrieve and modify these related pieces of data.

See *Model Data for Atomic Operations* (page 171) for an example data model that provides atomic updates for a single document.

Sharding

MongoDB uses *sharding* to provide horizontal scaling. These clusters support deployments with large data sets and high-throughput operations. Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` instances or *shards*.

To distribute data and application traffic in a sharded collection, MongoDB uses the *shard key* (page 687). Selecting the proper *shard key* (page 687) has significant implications for performance, and can enable or prevent query isolation and increased write capacity. It is important to consider carefully the field or fields to use as the shard key.

See *Sharding Introduction* (page 675) and *Shard Keys* (page 687) for more information.

Indexes

Use indexes to improve performance for common queries. Build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the `_id` field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes.
- Collections with high read-to-write ratio often benefit from additional indexes. Indexes do not affect un-indexed read operations.
- When active, each index consumes disk space and memory. This usage can be significant and should be tracked for capacity planning, especially for concerns over working set size.

See *Indexing Strategies* (page 551) for more information on indexes as well as *Analyze Query Performance* (page 117). Additionally, the MongoDB *database profiler* (page 239) may help identify inefficient queries.

Large Number of Collections

In certain situations, you might choose to store related information in several collections rather than in a single collection.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

```
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ...}
```

[6] Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple embedded documents within that single record are still atomic.

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 a large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.
- For each *database*, a single namespace file (i.e. `<database>.ns`) stores all meta-data for that database, and each index and collection has its own entry in the namespace file. MongoDB places `limits on the size of namespace files`.
- MongoDB has `limits on the number of namespaces`. You may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support. To get the current number of namespaces, run the following in the `mongo` shell:

  ```
  db.system.namespaces.count()
  ```

  The limit on the number of namespaces depend on the `<database>.ns` size. The namespace file defaults to 16 MB.

  To change the size of the *new* namespace file, start the server with the option `--nssize <new size MB>`. For existing databases, after starting up the server with `--nssize`, run the `db.repairDatabase()` command from the `mongo` shell. For impacts and considerations on running `db.repairDatabase()`, see `repairDatabase`.

Data Lifecycle Management

Data modeling decisions should take data lifecycle management into consideration.

The *Time to Live or TTL feature* (page 222) 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 219). Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and efficiently support operations that insert and read documents based on insertion order.

4.2.3 GridFS

On this page

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

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, [7] and stores each of those chunks as a separate document. By default GridFS limits chunk size to 255k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

[7] 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.

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 768).

Changed in version 2.4.10: The default chunk size changed from 256k to 255k.

Implement GridFS

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

- A MongoDB driver. See the `drivers` documentation for information on using GridFS with your driver.
- The `mongofiles` command-line tool in the `mongo` shell. See the `mongofiles` reference for complete documentation.

GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 183).
- `files` stores the file's metadata. For details, see *The files Collection* (page 183).

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 182).

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 182).

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` documentation for the specific behavior of your GridFS application. If your driver does not create this index, issue the following operation using the `mongo` shell:

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

Additional Resources

- Building MongoDB Applications with Binary Files Using GridFS: Part 1[8]
- Building MongoDB Applications with Binary Files Using GridFS: Part 2[9]

4.3 Data Model Examples and Patterns

The following documents provide overviews of various data modeling patterns and common schema design considerations:

Model Relationships Between Documents **(page 158)** Examples for modeling relationships between documents.

> ***Model One-to-One Relationships with Embedded Documents*** **(page 159)** Presents a data model that uses *embedded documents* (page 152) to describe one-to-one relationships between connected data.
>
> ***Model One-to-Many Relationships with Embedded Documents*** **(page 160)** Presents a data model that uses *embedded documents* (page 152) to describe one-to-many relationships between connected data.
>
> ***Model One-to-Many Relationships with Document References*** **(page 161)** Presents a data model that uses *references* (page 153) to describe one-to-many relationships between documents.

Model Tree Structures **(page 163)** Examples for modeling tree structures.

> ***Model Tree Structures with Parent References*** **(page 164)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 153) to "parent" nodes in "child" nodes.
>
> ***Model Tree Structures with Child References*** **(page 165)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 153) to "child" nodes in "parent" nodes.
>
> See *Model Tree Structures* (page 163) for additional examples of data models for tree structures.

Model Specific Application Contexts **(page 170)** Examples for models for specific application contexts.

> ***Model Data for Atomic Operations*** **(page 171)** Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.
>
> ***Model Data to Support Keyword Search*** **(page 172)** Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application's keyword search operations.

4.3.1 Model Relationships Between Documents

Model One-to-One Relationships with Embedded Documents **(page 159)** Presents a data model that uses *embedded documents* (page 152) to describe one-to-one relationships between connected data.

Model One-to-Many Relationships with Embedded Documents **(page 160)** Presents a data model that uses *embedded documents* (page 152) to describe one-to-many relationships between connected data.

Model One-to-Many Relationships with Document References **(page 161)** Presents a data model that uses *references* (page 153) to describe one-to-many relationships between documents.

[8] http://www.mongodb.com/blog/post/building-mongodb-applications-binary-files-using-gridfs-part-1?jmp=docs
[9] http://www.mongodb.com/blog/post/building-mongodb-applications-binary-files-using-gridfs-part-2?jmp=docs

Model One-to-One Relationships with Embedded Documents

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 152) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between `patron` and `address` data, the `address` belongs to the `patron`.

In the normalized data model, the `address` document contains a reference to the `patron` document.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "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.

Model One-to-Many Relationships with Embedded Documents

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 152) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between `patron` and `address` data, the `patron` has multiple `address` entities.

In the normalized data model, the `address` documents contain a reference to the `patron` document.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "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.

Model One-to-Many Relationships with Document References

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 153) between documents to describe relationships between connected data.

Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

```
{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

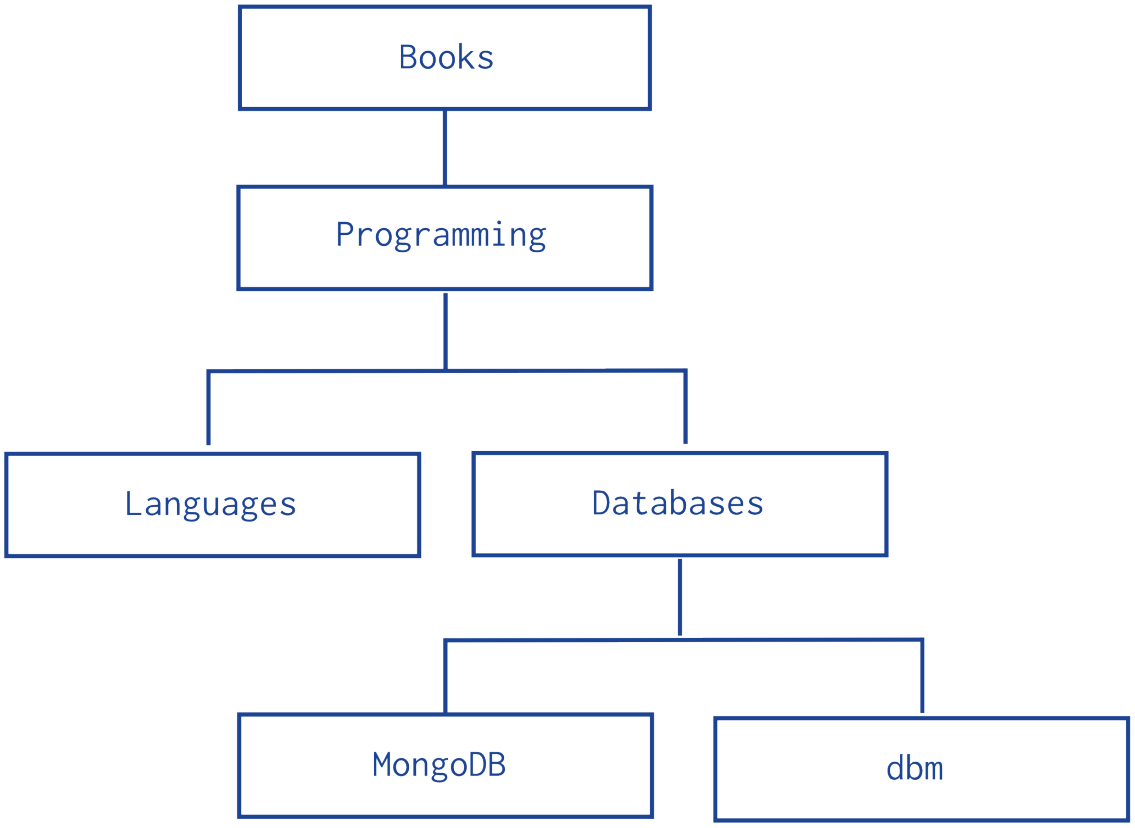
To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
```

```
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"
}
```

4.3.2 Model Tree Structures

MongoDB allows various ways to use tree data structures to model large hierarchical or nested data relationships.



Model Tree Structures with Parent References **(page 164)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 153) to "parent" nodes in "child" nodes.

Model Tree Structures with Child References **(page 165)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 153) to "child" nodes in "parent" nodes.

Model Tree Structures with an Array of Ancestors **(page 166)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 153) to "parent" nodes and an array that stores all ancestors.

Model Tree Structures with Materialized Paths **(page 168)** Presents a data model that organizes documents in a tree-like structure by storing full relationship paths between documents. In addition to the tree node, each document stores the `_id` of the nodes ancestors or path as a string.

Model Tree Structures with Nested Sets **(page 170)** Presents a data model that organizes documents in a tree-like structure using the *Nested Sets* pattern. This optimizes discovering subtrees at the expense of tree mutability.

Model Tree Structures with Parent References

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) 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 153) to "parent" nodes in children nodes.

Pattern

The *Parent References* pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node's parent.

Consider the following hierarchy of categories:

The following example models the tree using *Parent References*, storing the reference to the parent category in the field `parent`:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).parent
  ```

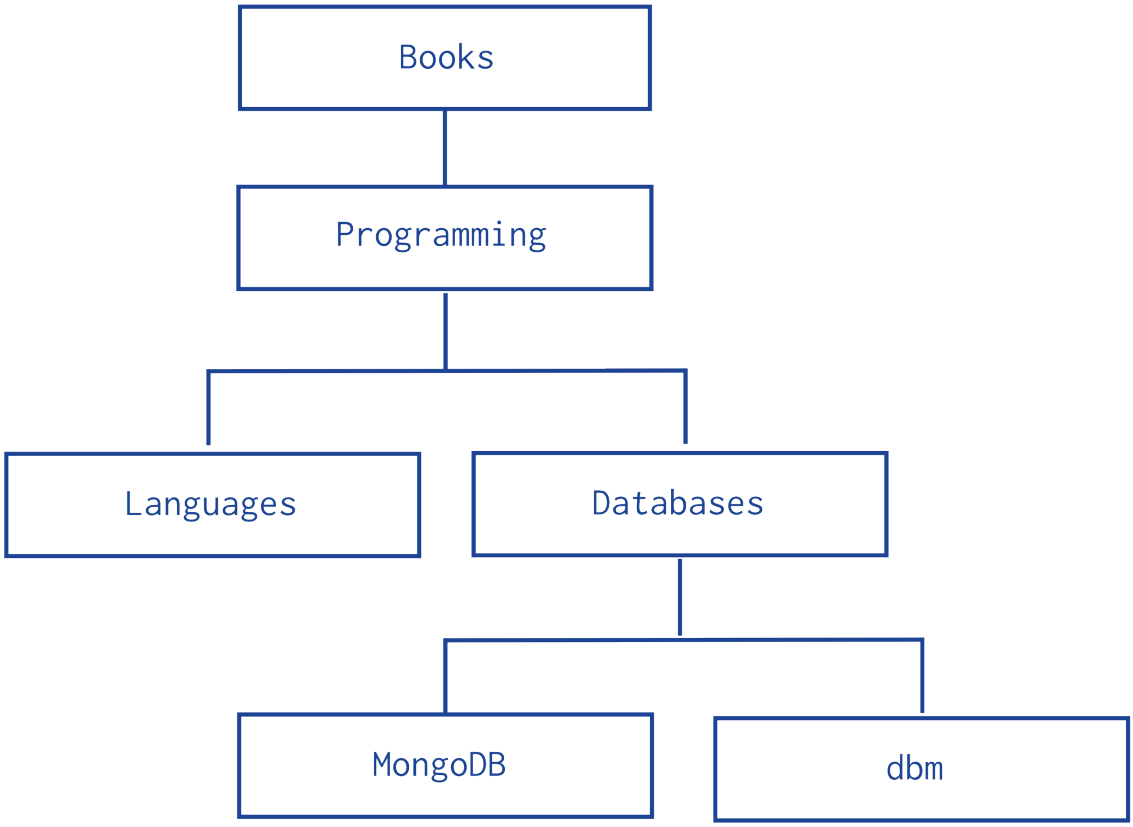
- You can create an index on the field `parent` to enable fast search by the parent node:

  ```
  db.categories.ensureIndex( { parent: 1 } )
  ```

- You can query by the `parent` field to find its immediate children nodes:

  ```
  db.categories.find( { parent: "Databases" } )
  ```

The *Parent Links* pattern provides a simple solution to tree storage but requires multiple queries to retrieve subtrees.



Model Tree Structures with Child References

On this page

Overview

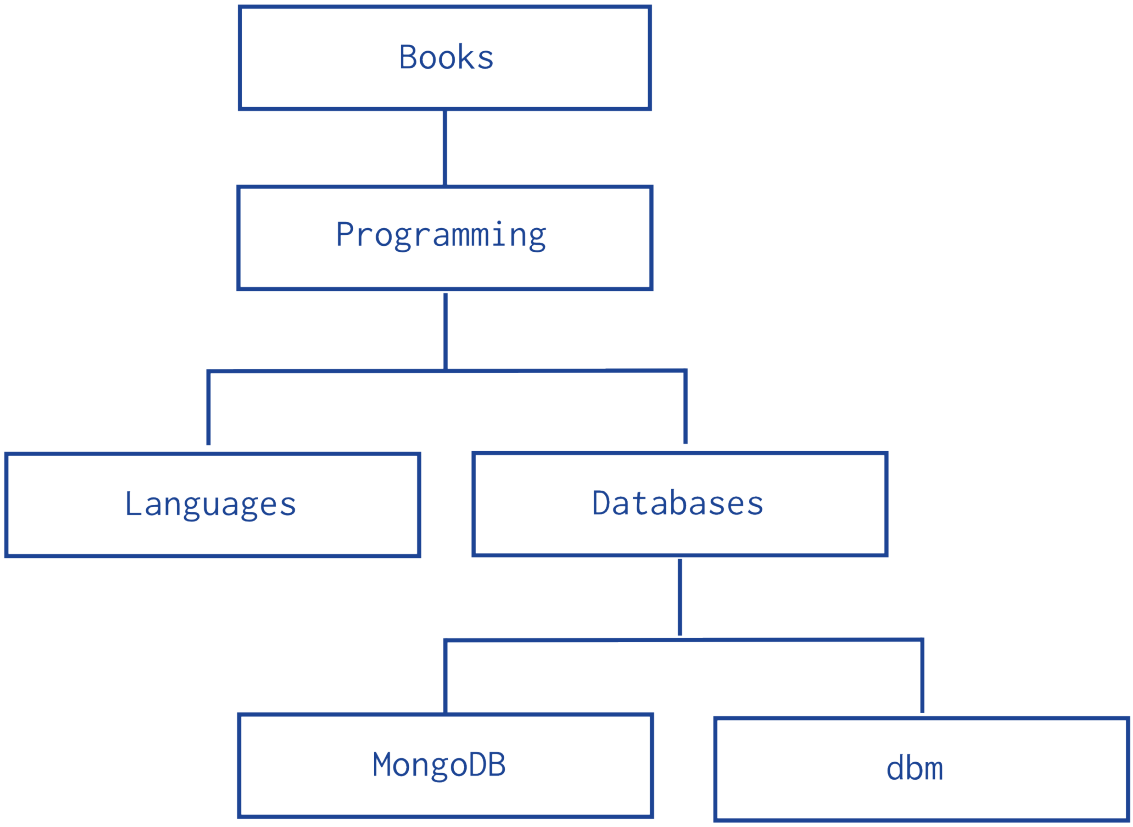
Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 153) in the parent-nodes to children nodes.

Pattern

The *Child References* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's children.

Consider the following hierarchy of categories:



The following example models the tree using *Child References*, storing the reference to the node's children in the field `children`:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
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.

Model Tree Structures with an Array of Ancestors

On this page

Overview

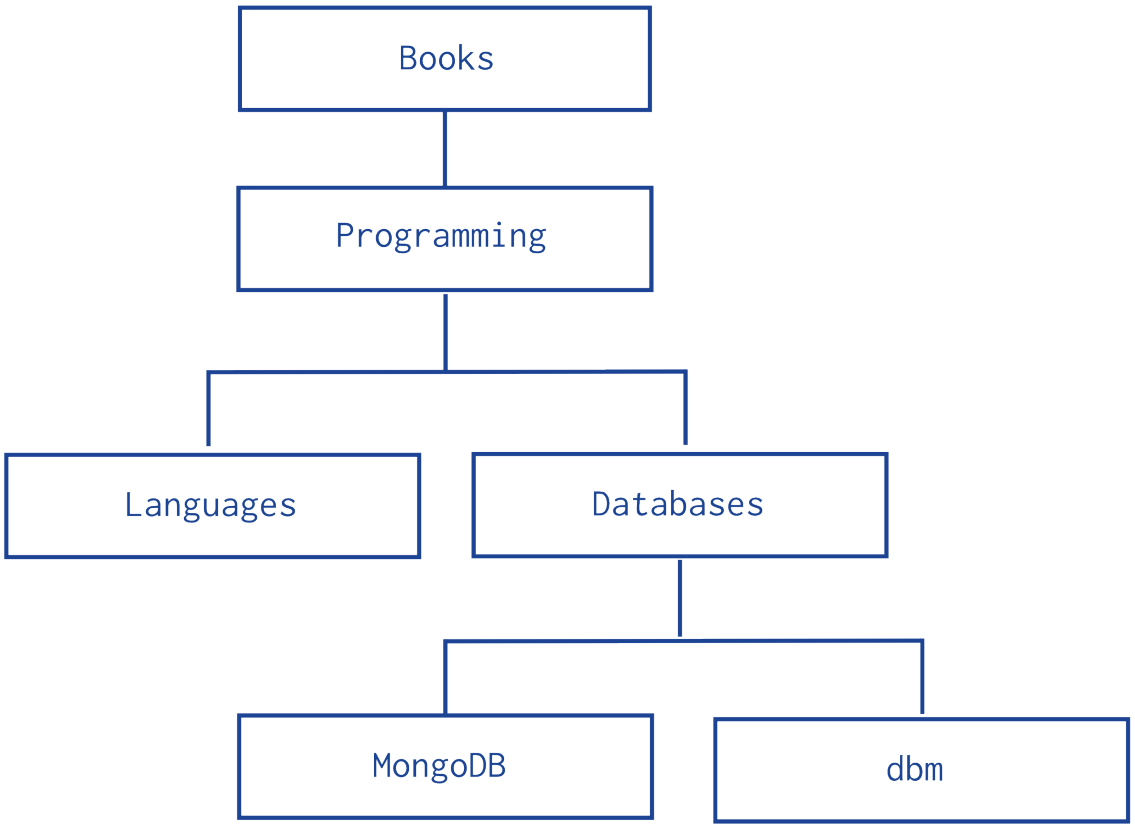
Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using *references* (page 153) to parent nodes and an array that stores all ancestors.

Pattern

The *Array of Ancestors* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's ancestors or path.

Consider the following hierarchy of categories:



The following example models the tree using *Array of Ancestors*. In addition to the `ancestors` field, these documents also store the reference to the immediate parent category in the `parent` field:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "
db.categories.insert( { _id: "dbm", ancestors: [ "Books", "Programming", "Databases" ], parent: "Data
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).ancestors
  ```

- You can create an index on the field `ancestors` to enable fast search by the ancestors nodes:

  ```
  db.categories.ensureIndex( { ancestors: 1 } )
  ```

- You can query by the field `ancestors` to find all its descendants:

  ```
  db.categories.find( { ancestors: "Programming" } )
  ```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the ancestors field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* (page 168) pattern but is more straightforward to use.

Model Tree Structures with Materialized Paths

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

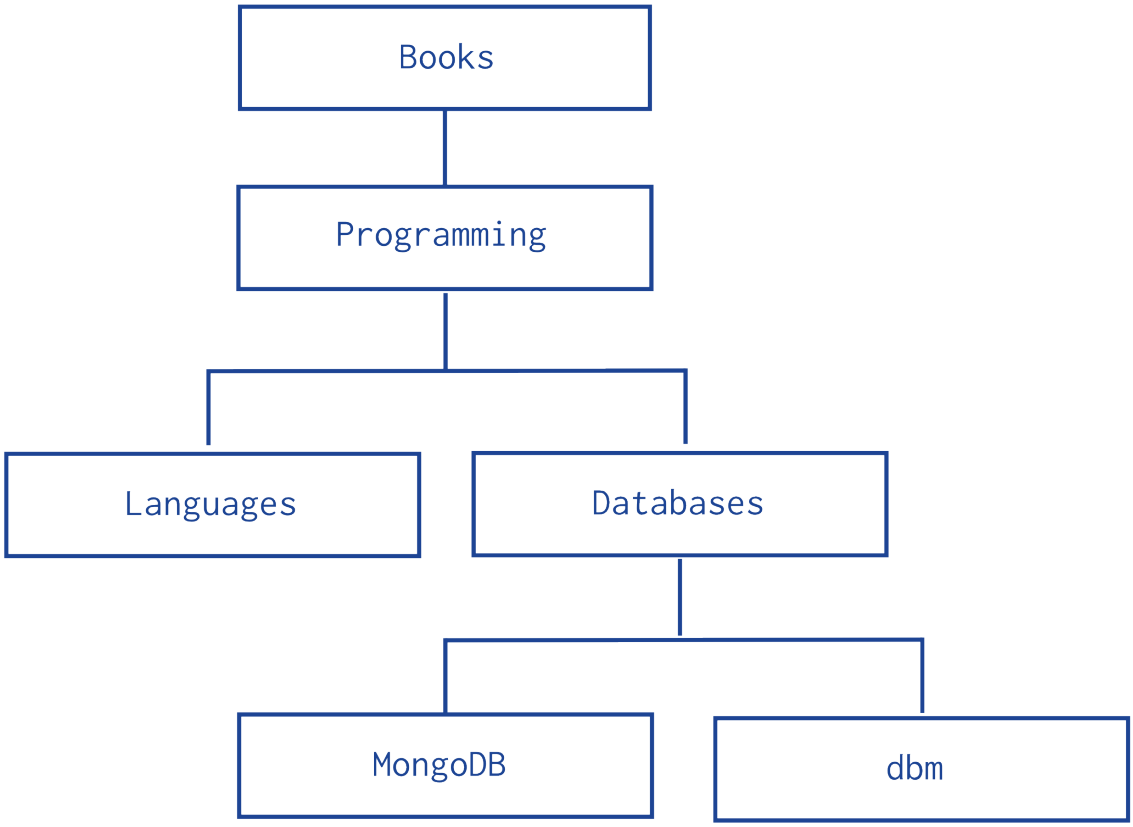
This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

Pattern

The *Materialized Paths* pattern stores each tree node in a document; in addition to the tree node, document stores as a string the id(s) of the node's ancestors or path. Although the *Materialized Paths* pattern requires additional steps of working with strings and regular expressions, the pattern also provides more flexibility in working with the path, such as finding nodes by partial paths.

Consider the following hierarchy of categories:

The following example models the tree using *Materialized Paths*, storing the path in the field `path`; the path string uses the comma `,` as a delimiter:




```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "dbm", path: ",Books,Programming,Databases," } )
```

- You can query to retrieve the whole tree, sorting by the field `path`:

  ```
  db.categories.find().sort( { path: 1 } )
  ```

- You can use regular expressions on the `path` field to find the descendants of `Programming`:

  ```
  db.categories.find( { path: /,Programming,/ } )
  ```

- You can also retrieve the descendants of `Books` where the `Books` is also at the topmost level of the hierarchy:

  ```
  db.categories.find( { path: /^,Books,/ } )
  ```

- To create an index on the field `path` use the following invocation:

  ```
  db.categories.ensureIndex( { path: 1 } )
  ```

  This index may improve performance depending on the query:

  - For queries from the root `Books` sub-tree (e.g. `http://docs.mongodb.org/manual/^,Books,/` or `http://docs.mongodb.org/manual/^,Books,Programming,/`), an index on the `path` field improves the query performance significantly.

– For queries of sub-trees where the path from the root is not provided in the query (e.g. `http://docs.mongodb.org/manual/,Databases,/`), or similar queries of sub-trees, where the node might be in the middle of the indexed string, the query must inspect the entire index.

  For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

Model Tree Structures with Nested Sets

On this page

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 151) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree like structure that optimizes discovering subtrees at the expense of tree mutability.

Pattern

The *Nested Sets* pattern identifies each node in the tree as stops in a round-trip traversal of the tree. The application visits each node in the tree twice; first during the initial trip, and second during the return trip. The *Nested Sets* pattern stores each tree node in a document; in addition to the tree node, document stores the id of node's parent, the node's initial stop in the `left` field, and its return stop in the `right` field.

Consider the following hierarchy of categories:

The following example models the tree using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "dbm", parent: "Databases", left: 8, right: 9 } )
```

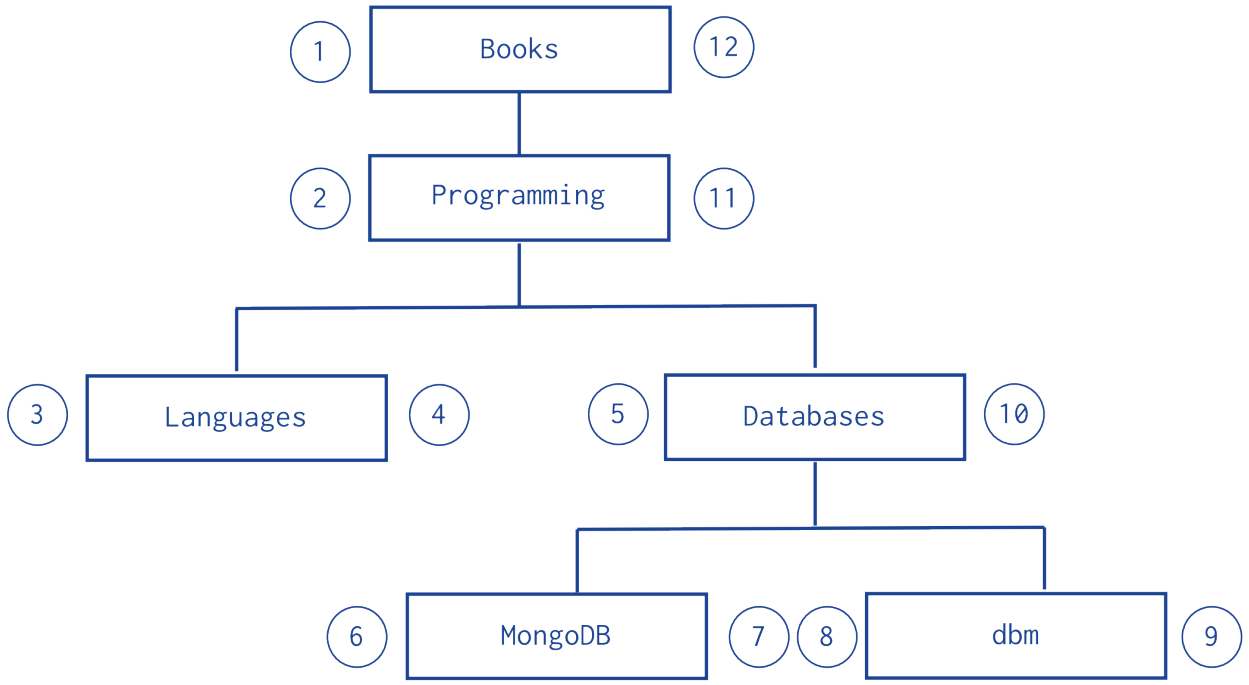
You can query to retrieve the descendants of a node:

```
var databaseCategory = db.categories.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } } )
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

4.3.3 Model Specific Application Contexts

Model Data for Atomic Operations **(page 171)** Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.



Model Data to Support Keyword Search **(page 172)** Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application's keyword search operations.

Model Monetary Data **(page 173)** Describes two methods to model monetary data in MongoDB.

Model Time Data **(page 175)** Describes how to deal with local time in MongoDB.

Model Data for Atomic Operations

On this page

Pattern

In MongoDB, write operations, e.g. `db.collection.update()`, `db.collection.findAndModify()`, `db.collection.remove()`, are atomic on the level of a single document. For fields that must be updated together, embedding the fields within the same document ensures that the fields can be updated atomically.

For example, consider a situation where you need to maintain information on books, including the number of copies available for checkout as well as the current checkout information.

The available copies of the book and the checkout information should be in sync. As such, embedding the `available` field and the `checkout` field within the same document ensures that you can update the two fields atomically.

```
{
    _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") } ]
}
```

Then to update with new checkout information, you can use the `db.collection.update()` method to atomically update both the `available` field and the `checkout` field:

```
db.books.update (
   { _id: 123456789, available: { $gt: 0 } },
   {
     $inc: { available: -1 },
     $push: { checkout: { by: "abc", date: new Date() } }
   }
)
```

The operation returns a `WriteResult()` object that contains information on the status of the operation:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The `nMatched` field shows that `1` document matched the update condition, and `nModified` shows that the operation updated `1` document.

If no document matched the update condition, then `nMatched` and `nModified` would be `0` and would indicate that you could not check out the book.

Model Data to Support Keyword Search

On this page

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 173) section for more information.

In 2.4, MongoDB provides a text search feature. See *Text Indexes* (page 501) for more information.

If your application needs to perform queries on the content of a field that holds text you can perform exact matches on the text or use `$regex` to use regular expression pattern matches. However, for many operations on text, these methods do not satisfy application requirements.

This pattern describes one method for supporting keyword search using MongoDB to support application search functionality, that uses keywords stored in an array in the same document as the text field. Combined with a *multi-key index* (page 491), this pattern can support application's keyword search operations.

Pattern

To add structures to your document to support keyword-based queries, create an array field in your documents and add the keywords as strings in the array. You can then create a *multi-key index* (page 491) on the array and create queries

that select values from the array.

Example

Given a collection of library volumes that you want to provide topic-based search. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the `Moby-Dick` volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

```
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and *multi-key indexes* (page 491); 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.

Model Monetary Data

On this page

Overview

MongoDB stores numeric data as either IEEE 754 standard 64-bit floating point numbers or as 32-bit or 64-bit signed integers. Applications that handle monetary data often require capturing fractional units of currency. However, arithmetic on floating point numbers, as implemented in modern hardware, often does not conform to requirements for monetary arithmetic. In addition, some fractional numeric quantities, such as one third and one tenth, have no exact representation in binary floating point numbers.

Note: Arithmetic mentioned on this page refers to server-side arithmetic performed by `mongod` or `mongos`, and not to client-side arithmetic.

This document describes two ways to model monetary data in MongoDB:

- *Exact Precision* (page 174) which multiplies the monetary value by a power of 10.
- *Arbitrary Precision* (page 175) which uses two fields for the value: one field to store the exact monetary value as a non-numeric and another field to store a floating point approximation of the value.

Use Cases for Exact Precision Model

If you regularly need to perform server-side arithmetic on monetary data, the exact precision model may be appropriate. For instance:

- If you need to query the database for exact, mathematically valid matches, use *Exact Precision* (page 174).
- If you need to be able to do server-side arithmetic, e.g., `$inc`, `$mul`, and `aggregation framework arithmetic`, use *Exact Precision* (page 174).

Use Cases for Arbitrary Precision Model

If there is no need to perform server-side arithmetic on monetary data, modeling monetary data using the arbitrary precision model may be suitable. For instance:

- If you need to handle arbitrary or unforeseen number of precision, see *Arbitrary Precision* (page 175).
- If server-side approximations are sufficient, possibly with client-side post-processing, see *Arbitrary Precision* (page 175).

Exact Precision

To model monetary data using the exact precision model:

1. Determine the maximum precision needed for the monetary value. For example, your application may require precision down to the tenth of one cent for monetary values in `USD` currency.

2. Convert the monetary value into an integer by multiplying the value by a power of 10 that ensures the maximum precision needed becomes the least significant digit of the integer. For example, if the required maximum precision is the tenth of one cent, multiply the monetary value by 1000.
3. Store the converted monetary value.

For example, the following scales `9.99 USD` by 1000 to preserve precision up to one tenth of a cent.

```
{ price: 9990, currency: "USD" }
```

The model assumes that for a given currency value:

- The scale factor is consistent for a currency; i.e. same scaling factor for a given currency.
- The scale factor is a constant and known property of the currency; i.e applications can determine the scale factor from the currency.

When using this model, applications must be consistent in performing the appropriate scaling of the values.

For use cases of this model, see *Use Cases for Exact Precision Model* (page 174).

Arbitrary Precision

To model monetary data using the arbitrary precision model, store the value in two fields:

1. In one field, encode the exact monetary value as a non-numeric data type; e.g., `BinData` or a `string`.
2. In the second field, store a double-precision floating point approximation of the exact value.

The following example uses the arbitrary precision model to store `9.99 USD` for the price and `0.25 USD` for the fee:

```
{
  price: { display: "9.99", approx: 9.9900000000000002, currency: "USD" },
  fee: { display: "0.25", approx: 0.2499999999999999, currency: "USD" }
}
```

With some care, applications can perform range and sort queries on the field with the numeric approximation. However, the use of the approximation field for the query and sort operations requires that applications perform client-side post-processing to decode the non-numeric representation of the exact value and then filter out the returned documents based on the exact monetary value.

For use cases of this model, see *Use Cases for Arbitrary Precision Model* (page 174).

Model Time Data

On this page

Overview

MongoDB *stores times in UTC* (page 189) by default, and will convert any local time representations into this form. Applications that must operate or report on some unmodified local time value may store the time zone alongside the UTC timestamp, and compute the original local time in their application logic.

Example

In the MongoDB shell, you can store both the current date and the current client's offset from UTC.

```
var now = new Date();
db.data.save( { date: now,
                offset: now.getTimezoneOffset() } );
```

You can reconstruct the original local time by applying the saved offset:

```
var record = db.data.findOne();
var localNow = new Date( record.date.getTime() -  ( record.offset * 60000 ) );
```

4.4 Data Model Reference

Documents **(page 176)** MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs.

Database References **(page 179)** Discusses manual references and DBRefs, which MongoDB can use to represent relationships between documents.

GridFS Reference **(page 182)** Convention for storing large files in a MongoDB Database.

ObjectId **(page 184)** A 12-byte BSON type that MongoDB uses as the default value for its documents' `_id` field if the `_id` field is not specified.

BSON Types **(page 186)** Outlines the unique *BSON* types used by MongoDB. See BSONspec.org[10] for the complete BSON specification.

4.4.1 Documents

On this page

MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs:

```
{ "item": "pencil", "qty": 500, "type": "no.2" }
```

Most user-accessible data structures in MongoDB are documents, including:

- All database records.
- *Query selectors* (page 64), which define what records to select for read, update, and delete operations.
- *Update definitions* (page 77), which define what fields to modify during an update.
- *Index specifications* (page 485), which define what fields to index.

[10] http://bsonspec.org/

- Data output by MongoDB for reporting and configuration, such as the output of the `serverStatus` and the *replica set configuration document* (page 660).

Document Format

MongoDB stores documents on disk in the *BSON* serialization format. BSON is a binary representation of *JSON* documents, though it contains more data types than JSON. For the BSON spec, see bsonspec.org[11]. See also *BSON Types* (page 186).

The `mongo` JavaScript shell and the `MongoDB language drivers` translate between BSON and the language-specific document representation.

Document Structure

MongoDB documents are composed of field-and-value pairs and have the following structure:

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

The value of a field can be any of the BSON *data types* (page 186), including other documents, arrays, and arrays of documents. The following document contains values of varying types:

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

The above fields have the following data types:

- `_id` holds an *ObjectId*.
- `name` holds an *embedded document* that contains the fields `first` and `last`.
- `birth` and `death` hold values of the *Date* type.
- `contribs` holds an *array of strings*.
- `views` holds a value of the *NumberLong* type.

Field Names

Field names are strings.

Documents (page 176) 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.

[11] http://bsonspec.org/

- The field names **cannot** start with the dollar sign (`$`) character.
- The field names **cannot** contain the dot (`.`) character.
- The field names **cannot** contain the `null` character.

BSON documents may have more than one field with the same name. Most `MongoDB interfaces`, however, represent MongoDB with a structure (e.g. a hash table) that does not support duplicate field names. If you need to manipulate documents that have more than one field with the same name, see the `driver documentation` for your driver.

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

Field Value Limit

For *indexed collections* (page 481), the values for the indexed fields have a `Maximum Index Key Length` limit. See `Maximum Index Key Length` for details.

Document Limitations

Documents have the following attributes:

Document Size Limit

The maximum BSON document size is 16 megabytes.

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

Document Field Order

MongoDB preserves the order of the document fields following write operations *except* for the following cases:

- The `_id` field is always the first field in the document.
- Updates that include `renaming` of field names may result in the reordering of fields in the document.

Changed in version 2.6: Starting in version 2.6, MongoDB actively attempts to preserve the field order in a document. Before version 2.6, MongoDB did not actively preserve the order of the fields in a document.

The `_id` Field

The `_id` field has the following behavior and constraints:

- By default, MongoDB creates a unique index on the `_id` field during the creation of a collection.
- The `_id` field is always the first field in the documents. If the server receives a document that does not have the `_id` field first, then the server will move the field to the beginning.
- The `_id` field may contain values of any *BSON data type* (page 186), other than an array.

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

The following are common options for storing values for `_id`:

- Use an *ObjectId* (page 184).
- Use a natural unique identifier, if available. This saves space and avoids an additional index.
- Generate an auto-incrementing number. See *Create an Auto-Incrementing Sequence Field* (page 130).
- 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[12] for information concerning UUID interoperability.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` will add the `_id` field and generate the `ObjectId`.

Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of an embedded document.

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

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

See also `$` positional operator for update operations and `$` projection operator when array index position is unknown.

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

```
'<embedded document>.<field>'
```

See also:

- *Embedded Documents* (page 102) for dot notation examples with embedded documents.
- *Arrays* (page 103) for dot notation examples with arrays.

4.4.2 Database References

On this page

[12] https://api.mongodb.org/

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:

- *Manual references* (page 180) 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 related data. These references are simple and sufficient for most use cases.
- *DBRefs* (page 181) are references from one document to another using the value of the first document's `_id` field, collection name, and, optionally, its database name. By including these names, DBRefs allow documents located in multiple collections to be more easily linked with documents from a single collection.

  To resolve DBRefs, your application must perform additional queries to return the referenced documents. Many `drivers` have helper methods that form the query for the DBRef automatically. The drivers [13] do not *automatically* resolve DBRefs into documents.

  DBRefs provide a common format and type to represent relationships among documents. The DBRef format also provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason to use DBRefs, use manual references instead.

Manual References

Background

Using manual references is 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.

[13] 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 180). 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 names. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using DBRefs.

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 name, 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 documents resemble the following document:

```
{ "$ref" : <value>, "$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.

Driver Support for DBRefs

C	The C driver contains no support for DBRefs. You can traverse references manually.
C++	The C++ driver contains no support for DBRefs. You can traverse references manually.
C#	The C# driver supports DBRefs using the MongoDBRef[14] class and `FetchDBRef` and `FetchDBRefAs` methods.
Haskell	The Haskell driver contains no support for DBRefs. You can traverse references manually.
Java	The DBRef[15] class provides support for DBRefs from Java.
JavaScript	The `mongo` shell's `JavaScript` interface provides a DBRef.
Node.js	The Node.js driver supports DBRefs using the DBRef[16] class and the dereference[17] method.
Perl	The Perl driver supports DBRefs using the MongoDB::DBRef[18] class. You can traverse references manually.
PHP	The PHP driver supports DBRefs, including the optional `$db` reference, using the MongoDBRef[19] class.
Python	The Python driver supports DBRefs using the DBRef[20] class and the dereference[21] method.
Ruby	The Ruby driver supports DBRefs using the DBRef[22] class and the dereference[23] method.
Scala	The Scala driver contains no support for DBRefs. You can traverse references manually.

Use

In most cases you should use the *manual reference* (page 180) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider using DBRefs.

4.4.3 GridFS Reference

On this page

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 183).
- `files` stores the file's metadata. For details, see *The files Collection* (page 183).

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.

[14] https://api.mongodb.org/csharp/current/html/T_MongoDB_Driver_MongoDBRef.htm
[15] https://api.mongodb.org/java/current/com/mongodb/DBRef.html
[16] http://mongodb.github.io/node-mongodb-native/api-bson-generated/db_ref.html
[17] http://mongodb.github.io/node-mongodb-native/api-generated/db.html#dereference
[18] https://metacpan.org/pod/MongoDB::DBRef
[19] http://www.php.net/manual/en/class.mongodbref.php/
[20] https://api.mongodb.org/python/current/api/bson/dbref.html
[21] https://api.mongodb.org/python/current/api/pymongo/database.html#pymongo.database.Database.deref eren ce
[22] https://api.mongodb.org/ruby/current/BSON/DBRef.html
[23] https://api.mongodb.org/ruby/current/Mongo/DB.html#dereference-instance_method

See also:

GridFS (page 156) for more information about GridFS.

The `chunks` Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

`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 157).

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 255 kilobytes.

Changed in version 2.4.10: The default chunk size changed from 256k to 255k.

`files.uploadDate`
The date the document was first stored by GridFS. This value has the `Date` type.

`files.md5`
An MD5 hash returned by the `filemd5` command. 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.

4.4.4 ObjectId

On this page

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*. MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified; i.e. if a document does not contain a top-level `_id` field, the MongoDB driver adds the `_id` field that holds an ObjectId. In addition, if the `mongod` receives a document to insert that does not contain an `_id` field, `mongod` will add the `_id` field that holds an ObjectId.

MongoDB clients should add an `_id` field with a unique ObjectId. Using ObjectIds for the `_id` field provides the following additional benefits:

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

Important: The relationship between the order of `ObjectId` values and generation time is not strict within a

single second. If multiple systems, or multiple processes or threads on a single system generate values, within a single second; `ObjectId` values do not represent a strict insertion order. Clock skew between clients can also result in non-strict ordering even for values because client drivers generate `ObjectId` values.

Also consider the *Documents* (page 176) section for related information on MongoDB's document orientation.

ObjectId()

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

- `str`

  The hexadecimal string representation of the object.

- `getTimestamp()`

  Returns the timestamp portion of the object as a Date.

- `toString()`

  Returns the JavaScript representation in the form of a string literal "`ObjectId(...)`".

  Changed in version 2.2: In previous versions `toString()` returns the hexadecimal string representation, which as of version 2.2 can be retrieved by the `str` property.

- `valueOf()`

  Returns the representation of the object as a hexadecimal string. The returned string is the `str` attribute.

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

Examples

Consider the following uses `ObjectId()` class in the `mongo` shell:

Generate a new ObjectId

To generate a new ObjectId, use the `ObjectId()` constructor with no argument:

```
x = ObjectId()
```

In this example, the value of `x` would be:

```
ObjectId("507f1f77bcf86cd799439011")
```

To generate a new ObjectId using the `ObjectId()` constructor with a unique hexadecimal string:

```
y = ObjectId("507f191e810c19729de860ea")
```

In this example, the value of `y` would be:

```
ObjectId("507f191e810c19729de860ea")
```

- To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

Convert an ObjectId into a Timestamp

To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

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

This operation will return the following Date object:

```
ISODate("2012-10-17T20:46:22Z")
```

Convert ObjectIds into Strings

Access the `str` attribute of an `ObjectId()` object, as follows:

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

This operation will return the following hexadecimal string:

```
507f191e810c19729de860ea
```

To return the hexadecimal string representation of an `ObjectId()`, use the `valueOf()` method as follows:

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

This operation returns the following output:

```
507f191e810c19729de860ea
```

To return the string representation of an `ObjectId()` object (in the form of a string literal `ObjectId(...)`), use the `toString()` method as follows:

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

This operation will return the following string output:

```
ObjectId("507f191e810c19729de860ea")
```

4.4.5 BSON Types

On this page

BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB. The BSON specification is located at bsonspec.org[24].

BSON supports the following data types as values in documents. Each data type has a corresponding number that can be used with the `$type` operator to query documents by BSON type.

[24] http://bsonspec.org/

Type	Number	Notes
Double	1	
String	2	
Object	3	
Array	4	
Binary data	5	
Undefined	6	Deprecated.
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	Query with `-1`.
Max key	127	

To determine a field's type, see *Check Types in the mongo Shell* (page 281).

If you convert BSON to JSON, see the `Extended JSON` reference.

Comparison/Sort Order

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)

MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose

value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

The following sections describe special considerations for particular BSON types.

ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation. Refer to the *ObjectId* (page 184) documentation for more information.

String

BSON strings are UTF-8. In general, drivers for each programming language convert from the language's string format to UTF-8 when serializing and deserializing BSON. This makes it possible to store most international characters in BSON strings with ease. [25] In addition, MongoDB `$regex` queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 189) type. Timestamp values are a 64 bit value where:

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

Within a single `mongod` instance, timestamp values are always unique.

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

Note: The BSON timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See *Date* (page 189) for more information.

If you insert a document containing an empty BSON timestamp in a top-level field, the MongoDB server will replace that empty timestamp with the current timestamp value. For example, if you create an insert a document with a timestamp value, as in the following operation:

```
var a = new Timestamp();

db.test.insert( { ts: a } );
```

Then, the `db.test.find()` operation will return a document that resembles the following:

```
{ "_id" : ObjectId("542c2b97bac0595474108b48"), "ts" : Timestamp(1412180887, 1) }
```

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

If `ts` were a field in an embedded document, the server would have left it as an empty timestamp value.

Changed in version 2.6: Previously, the server would only replace empty timestamp values in the first two fields, including `_id`, of an inserted document. Now MongoDB will replace any top-level field.

Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). This results in a representable date range of about 290 million years into the past and future.

The official BSON specification[26] refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed. [27] Negative values represent dates before 1970.

Example

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

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

Example

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

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

Example

Return the `Date` value as string:

```
mydate1.toString()
```

Example

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

```
mydate1.getMonth()
```

---

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

[27] 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.

CHAPTER 5

Administration

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

Administration Concepts **(page 191)** Core conceptual documentation of operational practices for managing MongoDB deployments and systems.

> ***MongoDB Backup Methods*** **(page 192)** Describes approaches and considerations for backing up a MongoDB database.
>
> ***Monitoring for MongoDB*** **(page 195)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.
>
> ***Production Notes*** **(page 210)** A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.
>
> Continue reading from *Administration Concepts* (page 191) for additional documentation of MongoDB administration.

Administration Tutorials **(page 231)** Tutorials that describe common administrative procedures and practices for operations for MongoDB instances and deployments.

> ***Configuration, Maintenance, and Analysis*** **(page 231)** Describes routine management operations, including configuration and performance analysis.
>
> ***Backup and Recovery*** **(page 256)** Outlines procedures for data backup and restoration with `mongod` instances and deployments.
>
> Continue reading from *Administration Tutorials* (page 231) for more tutorials of common MongoDB maintenance operations.

Administration Reference **(page 299)** Reference and documentation of internal mechanics of administrative features, systems and functions and operations.

See also:

The MongoDB Manual contains administrative documentation and tutorials though out several sections. See *Replica Set Tutorials* (page 606) and *Sharded Cluster Tutorials* (page 704) for additional tutorials and information.

5.1 Administration Concepts

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

Operational Strategies **(page 192)** Higher level documentation of key concepts for the operation and maintenance of MongoDB deployments, including backup, maintenance, and configuration.

MongoDB Backup Methods **(page 192)** Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB **(page 195)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration **(page 203)** Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Data Management **(page 217)** Core documentation that addresses issues in data management, organization, maintenance, and lifestyle management.

Data Center Awareness **(page 218)** Presents the MongoDB features that allow application developers and database administrators to configure their deployments to be more data center aware or allow operational and location-based separation.

Expire Data from Collections by Setting TTL **(page 222)** TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time.

Capped Collections **(page 219)** Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

Optimization Strategies for MongoDB **(page 223)** Techniques for optimizing application performance with MongoDB.

5.1.1 Operational Strategies

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

MongoDB Backup Methods **(page 192)** Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB **(page 195)** An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration **(page 203)** Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Import and Export MongoDB Data **(page 207)** Provides an overview of `mongoimport` and `mongoexport`, the tools MongoDB includes for importing and exporting data.

Production Notes **(page 210)** A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.

MongoDB Backup Methods

On this page

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

- *Backup by Copying Underlying Data Files* (page 193)
- *Backup with mongodump* (page 193)
- *MongoDB Cloud Manager Backup* (page 194)
- *Ops Manager Backup Software* (page 194)

Backup by Copying Underlying Data Files

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

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

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

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

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

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

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

For more information, see the *Backup and Restore with Filesystem Snapshots* (page 256) and *Backup a Sharded Cluster with Filesystem Snapshots* (page 267) for complete instructions on using LVM to create snapshots. Also see Back up and Restore Processes for MongoDB on Amazon EC2[1].

Backup with `mongodump`

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

`mongodump` and `mongorestore` can operate against a running `mongod` process, and can manipulate the underlying data files directly. By default, `mongodump` does not capture the contents of the *local database* (page 664).

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

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

[1] https://docs.mongodb.org/ecosystem/tutorial/backup-and-restore-mongodb-on-amazon-ec2

To mitigate the impact of `mongodump` on the performance of the replica set, use `mongodump` to capture backups from a *secondary* (page 569) member of a replica set. Alternatively, you can shut down a secondary and use `mongodump` with the data files directly. If you shut down a secondary to capture data with `mongodump` ensure that the operation can complete before its oplog becomes too stale to continue replicating.

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

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

See *Back Up and Restore with MongoDB Tools* (page 261), *Backup a Small Sharded Cluster with mongodump* (page 266), and *Backup a Sharded Cluster with Database Dumps* (page 269) for more information.

MongoDB Cloud Manager Backup

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

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

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

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

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

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

Ops Manager Backup Software

MongoDB Subscribers can install and run the same core software that powers *MongoDB Cloud Manager Backup* (page 194) on their own infrastructure. Ops Manager, an on-premise solution, has similar functionality to the cloud version and is available with Enterprise Advanced subscriptions.

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

Further Reading

Backup and Restore with Filesystem Snapshots **(page 256)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

Restore a Replica Set from MongoDB Backups **(page 260)** Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or MongoDB Cloud Manager[7] Backup file.

[2] https://cloud.mongodb.com/?jmp=docs
[3] https://cloud.mongodb.com/?jmp=docs
[4] https://docs.cloud.mongodb.com/
[5] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[6] https://docs.opsmanager.mongodb.com/current/
[7] https://cloud.mongodb.com/?jmp=docs

Back Up and Restore with MongoDB Tools **(page 261)** The procedure for writing the contents of a database to a BSON (i.e. binary) dump file for backing up MongoDB databases.

Backup and Restore Sharded Clusters **(page 265)** Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown **(page 274)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

Additional Resources

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

Monitoring for MongoDB

On this page

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

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

Note: MongoDB Cloud Manager[11] is a hosted service that provides monitoring, backup, and automated deployment of MongoDB instances. See MongoDB Cloud Manager[12] and the MongoDB Cloud Manager documentation[13] for more information.

Monitoring Strategies

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

- First, there is a set of utilities distributed with MongoDB that provides real-time reporting of database activities.
- Second, `database commands` return statistics regarding the current database state with greater fidelity.

[8] https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[9] http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs
[10] https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness
[11] https://cloud.mongodb.com/?jmp=docs
[12] https://cloud.mongodb.com/?jmp=docs
[13] https://docs.cloud.mongodb.com/

- Third, MongoDB Cloud Manager[14] collects data from running MongoDB deployments and provides visualization and alerts based on that data.

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

MongoDB Reporting Tools

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

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

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

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

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

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

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

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

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

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

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

`serverStatus` outputs an account of the state of a MongoDB instance. This command is rarely run directly. In most cases, the data is more meaningful when aggregated, as one would see with monitoring tools including MongoDB Cloud Manager[15]. Nevertheless, all administrators should be familiar with the data provided by `serverStatus`.

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

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

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

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

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

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

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

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

Tool	Plugin	Description
Ganglia[16]	mongodb-ganglia[17]	Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	gmond_python_modules[18]	Parses output from the `serverStatus` and `replSetGetStatus` commands.
Motop[19]	*None*	Realtime monitoring tool for MongoDB servers. Shows current operations ordered by durations every second.
mtop[20]	*None*	A top like tool.
Munin[21]	mongo-munin[22]	Retrieves server statistics.
Munin	mongomon[23]	Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).
Munin	munin-plugins Ubuntu PPA[24]	Some additional munin plugins not in the main distribution.
Nagios[25]	nagios-plugin-mongodb[26]	A simple Nagios check script, written in Python.

[16] http://sourceforge.net/apps/trac/ganglia/wiki
[17] https://github.com/quiiver/mongodb-ganglia
[18] https://github.com/ganglia/gmond_python_modules
[19] https://github.com/tart/motop
[20] https://github.com/beaufour/mtop
[21] http://munin-monitoring.org/
[22] https://github.com/erh/mongo-munin
[23] https://github.com/pcdummy/mongomon
[24] https://launchpad.net/ chris-lea/+archive/munin-plugins
[25] http://www.nagios.org/
[26] https://github.com/mzupan/nagios-plugin-mongodb

Also consider dex[27], an index and query analyzing tool for MongoDB that compares MongoDB log files and indexes to make indexing recommendations.

See also:

Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[28].

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

Name	Notes
MongoDB Cloud Manager[29]	MongoDB Cloud Manager is a cloud-based suite of services for managing MongoDB deployments. MongoDB Cloud Manager provides monitoring, backup, and automation functionality.
Scout[30]	Several plugins, including MongoDB Monitoring[31], MongoDB Slow Queries[32], and MongoDB Replica Set Monitoring[33].
Server Density[34]	Dashboard for MongoDB[35], MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.
Application Performance Management[36]	IBM has an Application Performance Management SaaS offering that includes monitor for MongoDB and other applications and middleware.

Process Logging

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

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

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

For example:

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

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

The following *database commands* also affect logging:

- `getLog`. Displays recent messages from the `mongod` process log.

[27] https://github.com/mongolab/dex
[28] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
[29] https://cloud.mongodb.com/?jmp=docs
[30] http://scoutapp.com
[31] https://scoutapp.com/plugin_urls/391-mongodb-monitoring
[32] http://scoutapp.com/plugin_urls/291-mongodb-slow-queries
[33] http://scoutapp.com/plugin_urls/2251-mongodb-replica-set-monitoring
[34] http://www.serverdensity.com
[35] http://www.serverdensity.com/mongodb-monitoring/
[36] http://ibmserviceengage.com

- `logRotate`. Rotates the log files for `mongod` processes only. See *Rotate Log Files* (page 243).

Diagnosing Performance Issues

Degraded performance in MongoDB is typically a function of the relationship between the quantity of data stored in the database, the amount of system RAM, the number of connections to the database, and the amount of time the database spends in a locked state.

In some cases performance issues may be transient and related to traffic load, data access patterns, or the availability of hardware on the host system for virtualized environments. Some users also experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. In other situations, performance issues may indicate that the database may be operating at capacity and that it is time to add additional capacity to the database.

The following are some causes of degraded performance in MongoDB.

Locks MongoDB uses a locking system to ensure data set consistency. However, if certain operations are long-running, or a queue forms, performance will slow as requests and operations wait for the lock. Lock-related slowdowns can be intermittent. To see if the lock has been affecting your performance, look to the data in the *globalLock* section of the `serverStatus` output. If `globalLock.currentQueue.total` is consistently high, then there is a chance that a large number of requests are waiting for a lock. This indicates a possible concurrency issue that may be affecting performance.

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

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 229) and disk reads.

Memory Usage MongoDB uses memory mapped files to store data. Given a data set of sufficient size, the MongoDB process will allocate all available memory on the system for its use. While this is part of the design, and affords MongoDB superior performance, the memory mapped files make it difficult to determine if the amount of RAM is sufficient for the data set.

The *memory usage statuses* metrics of the `serverStatus` output can provide insight into MongoDB's memory use. Check the resident memory use (i.e. `mem.resident`): if this exceeds the amount of system memory *and* there is a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

You should also check the amount of mapped memory (i.e. `mem.mapped`.) If this value is greater than the amount of system memory, some operations will require disk access *page faults* to read data from virtual memory and negatively affect performance.

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

Page faults triggered by MongoDB are reported as the total number of page faults in one second. To check for page faults, see the `extra_info.page_faults` value in the `serverStatus` output.

MongoDB on Windows counts both hard and soft page faults.

The MongoDB page fault counter may increase dramatically in moments of poor performance and may correlate with limited physical memory environments. Page faults also can increase while accessing much larger data sets, for example, scanning an entire collection. Limited and sporadic MongoDB page faults do not necessarily indicate a problem or a need to tune the database.

A single page fault completes quickly and is not problematic. However, in aggregate, large volumes of page faults typically indicate that MongoDB is reading too much data from disk. In many situations, MongoDB's read locks will "yield" after a page fault to allow other processes to read and avoid blocking while waiting for the next page to read into memory. This approach improves concurrency, and also improves overall throughput in high volume systems.

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

See *What are page faults?* (page 793) for more information.

Number of Connections In some cases, the number of connections between the application layer (i.e. clients) and the database can overwhelm the ability of the server to handle requests. This can produce performance irregularities. The following fields in the `serverStatus` document can provide insight:

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

If requests are high because there are numerous concurrent application requests, the database may have trouble keeping up with demand. If this is the case, then you will need to increase the capacity of your deployment. For read-heavy applications increase the size of your *replica set* and distribute read operations to *secondary* members. For write heavy applications, deploy *sharding* and add one or more *shards* to a *sharded cluster* to distribute load among `mongod` instances.

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

Unless constrained by system-wide limits MongoDB has no limit on incoming connections. You can modify system limits using the `ulimit` command, or by editing your system's `/etc/sysctl` file. See *UNIX ulimit Settings* (page 300) for more information.

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

The following profiling levels are available:

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

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

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

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

See

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

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

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

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

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

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

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

See also:

Optimization Strategies for MongoDB (page 223) addresses strategies that may improve the performance of your database queries and operations.

Replication and Monitoring

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

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

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

  By default, the oplog is 5 percent of total available disk space on 64-bit systems. For more information about changing the oplog size, see the *Change the Size of the Oplog* (page 634)

For causes of replication lag, see *Replication Lag* (page 654).

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

```
rs.status()
```

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

Sharding and Monitoring

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

See also:

See the *Sharding Concepts* (page 681) documentation for more information.

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

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

MongoDB Cloud Manager[37] monitors config servers and can create notifications if a config server becomes inaccessible. See the MongoDB Cloud Manager documentation[38] for more information.

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

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

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

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

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

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

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

Additional Resources

- MongoDB Production Readiness Consulting Package[39]

[37] https://cloud.mongodb.com/?jmp=docs
[38] https://docs.cloud.mongodb.com/
[39] https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness

Run-time Database Configuration

On this page

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

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

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

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

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

Configure the Database

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

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

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

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

```
logappend = true
journal = true
```

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

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

  New in version 2.6: `mongod` installed from official *.deb* (page 13) and *.rpm* (page 6) packages have the `bind_ip` configuration set to `127.0.0.1` by default.
- `port` is `27017`, which is the default MongoDB port for database instances. MongoDB can bind to any port. You can also filter access based on port using network filtering tools.

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

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

Security Considerations

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

In `YAML format`

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

Or, if using the older older configuration file format[40]:

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

[40] http://docs.mongodb.org/v2.4/reference/configuration-options

Consider the following explanation for these configuration decisions:

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

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

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

See also:

Security Concepts (page 316)

Replication and Sharding Configuration

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

In YAML format

```
replication:
   replSetName: set0
```

Or, if using the older configuration file format[41]:

```
replSet = set0
```

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

See also:

Replica set reconfiguration.

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

In YAML format

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

Or, if using the older configuration file format[42]:

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

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

See also:

The *Replica Set Security* (page 318) section for information on configuring authentication with replica sets.

The *Replication* (page 563) document for more information on replication in MongoDB and replica set configuration in general.

[41] http://docs.mongodb.org/v2.4/reference/configuration-options
[42] http://docs.mongodb.org/v2.4/reference/configuration-options

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

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

To set up one or three "config server" instances as *normal* (page 203) `mongod` instances, and then add the following configuration option:

In `YAML format`

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

Or, if using the older configuration file format[43]:

```
configsvr = true

bind_ip = 10.8.0.12
port = 27001
```

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

To set up shards, configure two or more `mongod` instance using your *base configuration* (page 203), with the `shardsvr` value for the `sharding.clusterRole` setting:

```
sharding:
   clusterRole: shardsvr
```

Or, if using the older configuration file format[44]:

```
shardsvr = true
```

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

In `YAML format`:

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

Or, if using the older configuration file format[45]:

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

Important: Always use 3 config servers in production environments.

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

In general, avoid modifying the `chunkSize` from the default value of 64, [46] and *should* ensure this setting is consistent among all `mongos` instances.

[43] http://docs.mongodb.org/v2.4/reference/configuration-options
[44] http://docs.mongodb.org/v2.4/reference/configuration-options
[45] http://docs.mongodb.org/v2.4/reference/configuration-options
[46] *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.

See also:

The *Sharding* (page 675) section of the manual for more information on sharding and cluster configuration.

Run Multiple Database Instances on the Same System

In many cases running multiple instances of `mongod` on a single system is not recommended. On some types of deployments [47] and for testing purposes you may need to run more than one `mongod` on a single system.

In these cases, use a *base configuration* (page 203) for each instance, but consider the following configuration values:

In `YAML format`:

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

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

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

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

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

Diagnostic Configurations

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

- `operationProfiling.mode` sets the *database profiler* (page 230) level. The profiler is not active by default because of the possible impact on the profiler itself on performance. Unless this setting is on, queries are not profiled.
- `operationProfiling.slowOpThresholdMs` configures the threshold which determines whether a query is "slow" for the purpose of the logging system and the *profiler* (page 230). The default value is 100 milliseconds. Set a lower value if the database profiler does not return useful results or a higher value to only log the longest running queries.
- `systemLog.verbosity` controls the amount of logging output that `mongod` write to the log. Only use this option if you are experiencing an issue that is not reflected in the normal logging level.

For more information, see also *Database Profiling* (page 230).

Import and Export MongoDB Data

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

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

On this page

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 192).

Warning: Because these tools primarily operate by interacting with a running `mongod` instance, they can impact the performance of your running database.
Not only do these processes create traffic for a running database instance, they also force the database to read all data through memory. When MongoDB reads infrequently used data, it can supplant more frequently accessed data, causing a deterioration in performance for the database's regular workload.

See also:

MongoDB Backup Methods (page 192) or MongoDB Cloud Manager Backup documentation[49] for more information on backing up MongoDB instances. Additionally, consider the following references for the MongoDB import/export tools:

- `mongoimport`
- `mongoexport`
- `mongorestore`
- `mongodump`

Data Import, Export, and Backup Operations

For resilient and non-disruptive backups, use a file system or block-level disk snapshot function, such as the methods described in the *MongoDB Backup Methods* (page 192) document. The tools and operations discussed provide functionality that is useful in the context of providing some kinds of backups.

In contrast, use import and export tools to backup a small subset of your data or to move data to or from a third 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.

Warning: `mongoimport` and `mongoexport` do not reliably preserve all rich *BSON* data types because *JSON* can only represent a subset of the types supported by BSON. As a result, data exported or imported with these tools may lose some measure of fidelity. See the `Extended JSON` reference for more information.

No matter how you decide to import or export your data, consider the following guidelines:

- Label files so that you can identify the contents of the export or backup as well as the point in time the export/backup reflect.
- Do not create or apply exports if the backup process itself will have an adverse effect on a production system.
- Make sure that 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.

[49] https://docs.cloud.mongodb.com/tutorial/nav/backup-use/

- Test backups and exports by restoring and importing to ensure that the backups are useful.

Human Intelligible Import/Export Formats

This section describes a process to import/export a collection to a file in a *JSON* or *CSV* format.

The examples in this section use the MongoDB tools `mongoimport` and `mongoexport`. These tools may also be useful for importing data into a MongoDB database from third party applications.

If you want to simply copy a database or collection from one instance to another, consider using the `copydb`, `clone`, or `cloneCollection` commands, which may be more suited to this task. The `mongo` shell provides the `db.copyDatabase()` method.

Collection Export with `mongoexport` You can use the `mongoexport` utility you can create a backup file.

> **Warning:** `mongoimport` and `mongoexport` do not reliably preserve all rich *BSON* data types because *JSON* can only represent a subset of the types supported by BSON. As a result, data exported or imported with these tools may lose some measure of fidelity. See the `Extended JSON` reference for more information.

In the most simple invocation, the command takes the following form:

```
mongoexport --collection collection --out collection.json
```

This will export all documents in the collection named `collection` into the file `collection.json`. Without the output specification (i.e. "*`--out collection.json`*"), `mongoexport` writes output to standard output (i.e. "stdout"). You can further narrow the results by supplying a query filter using the "*`--query`*" and limit results to a single database using the "*`--db`*" option. For instance:

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

This command returns all documents in the `sales` database's `contacts` collection, with a field named `field` with a value of `1`. Enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment. The resulting documents will return on standard output.

By default, `mongoexport` returns one *JSON document* per MongoDB document. Specify the "*`--jsonArray`*" argument to return the export as a single *JSON* array. Use the "*`--csv`*" file to return the result in CSV (comma separated values) format.

If your `mongod` instance is not running, you can use the "*`--dbpath`*" option to specify the location to your MongoDB instance's database files. See the following example:

```
mongoexport --db sales --collection contacts --dbpath /srv/MongoDB/
```

This reads the data files directly. This locks the data directory to prevent conflicting writes. The `mongod` process must *not* be running or attached to these data files when you run `mongoexport` in this configuration.

The "*`--host`*" and "*`--port`*" options allow you to specify a non-local host to connect to capture the export. Consider the following example:

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

On any `mongoexport` command you may, as above specify username and password credentials as above.

Collection Import with `mongoimport` To restore a backup taken with `mongoexport`. Most of the arguments to `mongoexport` also exist for `mongoimport`.

Warning: `mongoimport` and `mongoexport` do not reliably preserve all rich *BSON* data types because *JSON* can only represent a subset of the types supported by BSON. As a result, data exported or imported with these tools may lose some measure of fidelity. See the `Extended JSON` reference for more information.

Consider the following command:

```
mongoimport --collection collection --file collection.json
```

This imports the contents of the file `collection.json` into the collection named `collection`. If you do not specify a file with the "*`--file`*" option, `mongoimport` accepts input over standard input (e.g. "stdin.")

If you specify the "*`--upsert`*" option, all of `mongoimport` operations will attempt to update existing documents in the database and insert other documents. This option will cause some performance impact depending on your configuration.

You can specify the database option *`--db`* to import these documents to a particular database. If your MongoDB instance is not running, use the "*`--dbpath`*" option to specify the location of your MongoDB instance's database files. Consider using the "*`--journal`*" option to ensure that `mongoimport` records its operations in the journal. The `mongod` process must *not* be running or attached to these data files when you run `mongoimport` in this configuration.

Use the "*`--ignoreBlanks`*" option to ignore blank fields. For *CSV* and *TSV* imports, this option provides the desired functionality in most cases: it avoids inserting blank fields in MongoDB documents.

Production Notes

On this page

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

Note: MongoDB Cloud Manager[50] is a hosted service that provides monitoring, backup, and automated deployment of MongoDB instances. See MongoDB Cloud Manager[51] and the MongoDB Cloud Manager documentation[52] for more information.

Packages

[50] https://cloud.mongodb.com/?jmp=docs
[51] https://cloud.mongodb.com/?jmp=docs
[52] https://docs.cloud.mongodb.com/

MongoDB Be sure you have the latest stable release. All releases are available on the Downloads[53] page. This is a good place to verify what is current, even if you then choose to install via a package manager.

Always use 64-bit builds for production. The 32-bit build MongoDB offers for test and development environments is not suitable for production deployments as it can store no more than 2GB of data. See the *32-bit limitations page* (page 764) for more information.

32-bit builds exist to support use on development machines.

Operating Systems MongoDB distributions are currently available for Mac OS X, Linux, Windows Server 2008 R2 64bit, Windows 7 (32 bit and 64 bit), Windows Vista, and Solaris platforms.

Note: MongoDB uses the GNU C Library[54] (glibc) if available on a system. MongoDB requires version at least `glibc-2.12-1.2.el6` to avoid a known bug with earlier versions. For best results use at least version 2.13.

Concurrency

In earlier versions of MongoDB, all write operations contended for a single readers-writer lock on the MongoDB instance. As of version 2.2, each database has a readers-writer lock that allows concurrent reads access to a database, but gives exclusive access to a single write operation per database. See the *Concurrency* (page 777) page for more information.

Journaling

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee that MongoDB is able to quickly recover the *write operations* (page 77) following a crash or other serious failure.

In order to ensure that `mongod` will be able to recover its data files and keep the data files in a valid state following a crash, leave journaling enabled. See *Journaling* (page 309) for more information.

Networking

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

Note: By default, `authorization` is not enabled and `mongod` assumes a trusted environment. You can enable *security/auth* (page 316) mode if you need it.

See documents in the *Security Section* (page 313) for additional information, specifically:

- *Configuration Options* (page 323)
- *Firewalls* (page 324)
- *Network Security Tutorials* (page 330)

For Windows users, consider the Windows Server Technet Article on TCP Configuration[55] when deploying MongoDB on Windows.

[53] http://www.mongodb.org/downloads
[54] http://www.gnu.org/software/libc/
[55] http://technet.microsoft.com/en-us/library/dd349797.aspx

Connection Pools To avoid overloading the connection resources of a single `mongod` or `mongos` instance, ensure that clients maintain reasonable connection pool sizes.

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

Hardware Considerations

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

Hardware Requirements and Limitations The hardware for the most effective MongoDB deployments have the following properties:

Allocate Sufficient RAM and CPU As with all software, more RAM and a faster CPU clock speed are important for performance.

In general, databases are not CPU bound. As such, increasing the number of cores can help, but does not provide significant marginal return.

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

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

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

Avoid Remote File Systems

- Remote file storage can create performance problems in MongoDB. See *Remote Filesystems* (page 213) for more information about storage and MongoDB.

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

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

See also:

- The MySQL "swap insanity" problem and the effects of NUMA[56] post, which describes the effects of NUMA on databases. The post introduces NUMA and its goals, and illustrates how these goals are not compatible with production databases. Although the blog post addresses the impact of NUMA for MySQL, the issues for MongoDB are similar.

[56] http://jcole.us/blog/archives/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/

- NUMA: An Overview[57].

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

Configuring NUMA on Linux When running MongoDB on Linux, you may instead use the `numactl` command and start the MongoDB programs (`mongod`, including the *config servers* (page 684); `mongos`; or clients) in the following manner:

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

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

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

To fully disable NUMA behavior, you must perform both operations. For more information, see the Documentation for /proc/sys/vm/*[58].

Disk and Storage Systems

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

The method `mongod` uses to map memory files to memory ensures that the operating system will never store MongoDB data in swap space. On Windows systems, MongoDB requires extra swap space due to commitment limits. For details, see *MongoDB on Windows* (page 216).

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

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

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

Remote Filesystems The Network File System protocol (NFS) is not recommended for use with MongoDB as some versions perform poorly.

Performance problems arise when both the data files and the journal files are hosted on NFS. You may experience better performance if you place the journal on local or `iscsi` volumes. If you must use NFS, add the following NFS options to your `/etc/fstab` file: `bg`, `nolock`, and `noatime`.

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

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

---

[57] https://queue.acm.org/detail.cfm?id=2513149
[58] http://www.kernel.org/doc/Documentation/sysctl/vm.txt

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

Architecture

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

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

See the *Write Concern* (page 82) document for more information about choosing an appropriate write concern level for your deployment.

Replica Sets See the *Replica Set Architectures* (page 575) document for an overview of architectural considerations for replica set deployments.

Sharded Clusters See the *Sharded Cluster Production Architecture* (page 686) document for an overview of recommended sharded cluster architectures for production deployments.

Platforms

MongoDB on Linux

Important: The following discussion only applies to Linux, and therefore does not affect deployments where `mongod` instances run other UNIX-like systems or on Windows.

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

MongoDB preallocates its database files before using them and often creates large files. As such, you should use the Ext4 and XFS file systems:

- In general, if you use the Ext4 file system, use at least version `2.6.23` of the Linux Kernel.
- In general, if you use the XFS file system, use at least version `2.6.25` of the Linux Kernel.
- Some Linux distributions require different versions of the kernel to support using ext4 and/or xfs:

Linux Distribution	Filesystem	Kernel Version
CentOS 5.5	ext4, xfs	`2.6.18-194.el5`
CentOS 5.6	ext4, xfs	`2.6.18-238.el5`
CentOS 5.8	ext4, xfs	`2.6.18-308.8.2.el5`
CentOS 6.1	ext4, xfs	`2.6.32-131.0.15.el6.x86_64`
RHEL 5.6	ext4	`2.6.18-238`
RHEL 6.0	xfs	`2.6.32-71`
Ubuntu 10.04.4 LTS	ext4, xfs	`2.6.32-38-server`
Amazon Linux AMI release 2012.03	ext4	`3.2.12-3.2.4.amzn1.x86_64`

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

Recommended Configuration

- Turn off `atime` for the storage volume containing the *database files*.
- Set the file descriptor limit, `-n`, and the user process limit (ulimit), `-u`, above 20,000, according to the suggestions in the *ulimit* (page 300) document. A low ulimit will affect MongoDB when under heavy use and can produce errors and lead to failed connections to MongoDB processes and loss of service.
- Disable Transparent Huge Pages, as MongoDB performs better with normal (4096 bytes) virtual memory pages. See *Transparent Huge Pages Settings* (page 232).
- Disable NUMA in your BIOS. If that is not possible see *MongoDB on NUMA Hardware* (page 212).
- Configure SELinux on Red Hat. For more information, see *Configure SELinux for MongoDB* (page 8) and *Configure SELinux for MongoDB Enterprise* (page 30).
- Ensure that readahead settings for the block devices that store the database files are appropriate. For random access use patterns, set low readahead values. A readahead of 32 (16kb) often works well.

  For a standard block device, you can run `sudo blockdev --report` to get the readahead settings and `sudo blockdev --setra <value> <device>` to change the readahead settings. Refer to your specific operating system manual for more information.
- Use the Network Time Protocol (NTP) to synchronize time among your hosts. This is especially important in sharded clusters.

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

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

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

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

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

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

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

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

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

MongoDB on Windows

Install Hotfix Microsoft has released a hotfix for Windows 7 and Windows Server 2008 R2, KB2731284[59], that repairs a bug in these operating systems' use of memory-mapped files that adversely affects the performance of MongoDB.

Install this hotfix to obtain significant performance improvements on MongoDB 2.6.6 and later releases in the 2.6 series.

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

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

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

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

MongoDB on Virtual Environments The section describes considerations when running MongoDB in some of the more common virtual environments.

For all platforms, consider *Scheduling for Virtual Devices* (page 214).

EC2 MongoDB is compatible with EC2.

MongoDB Cloud Manager[60] provides integration with Amazon Web Services (AWS) and lets you deploy new EC2 instances directly from MongoDB Cloud Manager. See Configure AWS Integration[61] for more details.

VMWare MongoDB is compatible with VMWare. As some users have run into issues with VMWare's memory overcommit feature, disabling the feature is recommended.

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

OpenVZ Some users have had issues when running MongoDB on some older version of OpenVZ due to its handling of virtual memory, as with VMWare.

This issue seems to have been resolved in the more recent versions of OpenVZ.

Performance Monitoring

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

[59] http://support.microsoft.com/kb/2731284
[60] https://cloud.mongodb.com/?jmp=docs
[61] https://docs.cloud.mongodb.com/tutorial/configure-aws-settings/

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

```
iostat -xmt 1
```

Key fields from `iostat`:

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

bwm-ng bwm-ng[62] is a command-line tool for monitoring network use. If you suspect a network-based bottleneck, you may use `bwm-ng` to begin your diagnostic process.

Backups

To make backups of your MongoDB database, please refer to *MongoDB Backup Methods Overview* (page 192).

Additional Resources

- Blog Post: Capacity Planning and Hardware Provisioning for MongoDB In Ten Minutes[63]
- Whitepaper: MongoDB Multi-Data Center Deployments[64]
- Whitepaper: Security Architecture[65]
- Whitepaper: MongoDB Architecture Guide[66]
- Presentation: MongoDB Administration 101[67]
- MongoDB Production Readiness Consulting Package[68]

5.1.2 Data Management

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

Data Center Awareness **(page 218)** Presents the MongoDB features that allow application developers and database administrators to configure their deployments to be more data center aware or allow operational and location-based separation.

Capped Collections **(page 219)** Capped collections provide a special type of size-constrained collections that preserve insertion order and can support high volume inserts.

Expire Data from Collections by Setting TTL **(page 222)** TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time.

[62] http://www.gropp.org/?id=projects&sub=bwm-ng
[63] https://www.mongodb.com/blog/post/capacity-planning-and-hardware-provisioning-mongodb-ten-minutes?jmp=docs
[64] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
[65] https://www.mongodb.com/lp/white-paper/mongodb-security-architecture?jmp=docs
[66] https://www.mongodb.com/lp/whitepaper/architecture-guide?jmp=docs
[67] http://www.mongodb.com/presentations/webinar-mongodb-administration-101?jmp=docs
[68] https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness

Data Center Awareness

On this page

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

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

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

Operational Segregation in MongoDB Deployments **(page 218)** MongoDB lets you specify that certain application operations use certain `mongod` instances.

Tag Aware Sharding **(page 746)** Tags associate specific ranges of *shard key* values with specific shards for use in managing deployment patterns.

Manage Shard Tags **(page 747)** Use tags to associate specific ranges of shard key values with specific shards.

Operational Segregation in MongoDB Deployments

On this page

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

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

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

Specifically, with MongoDB, you can:

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

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

- *Read Preferences* (page 591), which controls how drivers help applications target read operations to members of a replica set.
- *Write Concerns* (page 82), which controls how MongoDB ensures that write operations propagate to members of a replica set.
- *Replica Set Tags* (page 641), 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 746), 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 563), and *sharding* (page 675).

Additional Resource MongoDB Multi-Data Center Deployments Whitepaper[69]

Further Reading

- The *Write Concern* (page 82) and *Read Preference* (page 591) documents, which address capabilities related to data center awareness.
- *Deploy a Geographically Redundant Replica Set* (page 612).

Additional Resource

MongoDB Multi-Data Center Deployments Whitepaper[70]

Capped Collections

On this page

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

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

Capped collections have the following behaviors:

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

[69] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
[70] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs

- Capped collections guarantee that insertion order is identical to the order on disk (*natural order*) and do so by prohibiting updates that increase document size. Capped collections only allow updates that fit the original document size, which ensures a document does not change its location on disk.
- Capped collections automatically remove the oldest documents in the collection without requiring scripts or explicit remove operations.

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

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

Recommendations and Restrictions

- You can only make in-place updates of documents. 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.

- If you update a document in a capped collection to a size smaller than its original size, and then a secondary resyncs from the primary, the secondary will replicate and allocate space based on the current smaller document size. If the primary then receives an update which increases the document back to its original size, the primary will accept the update but the secondary will fail with a `failing update: objects in a capped ns cannot grow` error message.

  To prevent this error, create your secondary from a snapshot of one of the other up-to-date members of the replica set. Follow *our tutorial on filesystem snapshots* (page 256) to seed your new secondary.

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

- You cannot delete documents from a capped collection. To remove all documents from a collection, use the `drop()` method to drop the collection.
- You cannot shard a capped collection.
- Capped collections created after 2.2 have an `_id` field and an index on the `_id` field by default. Capped collections created before 2.2 do not have an index on the `_id` field by default. If you are using capped collections with replication prior to 2.2, you should explicitly create an index on the `_id` field.

  Warning: If you have a capped collection in a *replica set* outside of the `local` database, before 2.2, you should create a unique index on `_id`. Ensure uniqueness using the `unique: true` option to the `ensureIndex()` method or by using an *ObjectId* for the `_id` field. Alternately, you can use the `autoIndexId` option to `create` when creating the capped collection, as in the *Query a Capped Collection* (page 221) 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.
- The aggregation pipeline operator `$out` cannot write results to a capped collection.

Procedures

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

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

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

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

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

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

See

`createCollection()` and `create`.

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

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

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

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

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

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

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

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

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

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

Automatically Remove Data After a Specified Period of Time For additional flexibility when expiring data, consider MongoDB's *TTL* indexes, as described in *Expire Data from Collections by Setting TTL* (page 222). 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 222) 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 128) for information on creating a tailable cursor.

Expire Data from Collections by Setting TTL

On this page

New in version 2.2.

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

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

A special *TTL index property* (page 504) supports the implementation of TTL collections. The TTL feature relies on a background thread in `mongod` that reads the date-typed values in the index and removes expired *documents* from the collection.

Procedures

To create a *TTL index* (page 504), use the `db.collection.ensureIndex()` method with the `expireAfterSeconds` option on a field whose value is either a *date* (page 189) or an array that contains *date values* (page 189).

Note: The TTL index is a single field index. Compound indexes do not support the TTL property. For more information on TTL indexes, see *TTL Indexes* (page 504).

Expire Documents after a Specified Number of Seconds To expire data after a specified number of seconds has passed since the indexed field, create a TTL index on a field that holds values of BSON date type or an array of BSON date-typed objects *and* specify a positive non-zero value in the `expireAfterSeconds` field. A document will expire when the number of seconds in the `expireAfterSeconds` field has passed since the time specified in its indexed field. [71]

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

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

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

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

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

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

See also:

`$currentDate` operator

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

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

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

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

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

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

5.1.3 Optimization Strategies for MongoDB

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

This section describes techniques for optimizing application performance with MongoDB.

Evaluate Performance of Current Operations **(page 224)** MongoDB provides introspection tools that describe the query execution process, to allow users to test queries and build more efficient queries.

Optimize Query Performance **(page 224)** Introduces the use of *projections* (page 67) to reduce the amount of data MongoDB sends to clients.

Design Notes **(page 226)** A collection of notes related to the architecture, design, and administration of MongoDB-based applications.

Evaluate Performance of Current Operations

On this page

The following sections describe techniques for evaluating operational performance.

Use the Database Profiler to Evaluate Operations Against the Database

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

For more information, see *Database Profiling* (page 230).

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

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

Use `$explain` to Evaluate Query Performance

The `explain()` method returns statistics on a query, and reports the index MongoDB selected to fulfill the query, as well as information about the internal operation of the query.

Example

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

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

See *Analyze Query Performance* (page 117) for more details.

Optimize Query Performance

On this page

Create Indexes to Support Queries

For commonly issued queries, create *indexes* (page 481). If a query searches multiple fields, create a *compound index* (page 489). 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 441).

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

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

Limit the Number of Query Results to Reduce Network Demand

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

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

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

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

Use Projections to Return Only Necessary Data

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

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

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

For more information on using projections, see *Limit Fields to Return from a Query* (page 112).

Use `$hint` to Select a Particular Index

In most cases the *query optimizer* (page 72) selects the optimal index for a specific operation; however, you can force MongoDB to use a specific index using the `hint()` method. Use `hint()` to support performance testing, or on some queries where you must select a field or field included in several indexes.

Use the Increment Operator to Perform Operations Server-Side

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

Design Notes

On this page

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

Schema Considerations

Dynamic Schema Data in MongoDB has a *dynamic schema*. *Collections* do not enforce *document* structure. This facilitates iterative development and polymorphism. Nevertheless, collections often hold documents with highly homogeneous structures. See *Data Modeling Concepts* (page 151) for more information.

Some operational considerations include:

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

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

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

Consider:

- storing data in a normalized case format, or
- using regular expressions ending with the `i` option, and/or
- using `$toLower` or `$toUpper` in the *aggregation framework* (page 439).

Type Sensitive Fields MongoDB data is stored in the BSON format, a binary encoded serialization of JSON-like documents. BSON encodes additional type information. See bsonspec.org[72] for more information.

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

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

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

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

General Considerations

By Default, Updates Affect one Document To update multiple documents that meet your query criteria, set the `update multi` option to `true` or `1`. See: *Update Multiple Documents* (page 80).

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

BSON Document Size Limit The `BSON Document Size` limit is currently set at 16MB per document. If you require larger documents, use *GridFS* (page 156).

No Fully Generalized Transactions MongoDB does not have *fully generalized transactions* (page 86). If you model your data using rich documents that closely resemble your application's objects, each logical object will be in one MongoDB document. MongoDB allows you to modify a document in a single atomic operation. These kinds of data modification pattern covers most common uses of transactions in other systems.

Replica Set Considerations

Use an Odd Number of Replica Set Members *Replica sets* (page 563) perform consensus elections. To ensure that elections will proceed successfully, either use an odd number of members, typically three, or else use an *arbiter* to ensure an odd number of votes.

Keep Replica Set Members Up-to-Date MongoDB replica sets support *automatic failover* (page 583). It is important for your secondaries to be up-to-date. There are various strategies for assessing consistency:

1. Use monitoring tools to alert you to lag events. See *Monitoring for MongoDB* (page 195) for a detailed discussion of MongoDB's monitoring options.
2. Specify appropriate write concern.

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

3. If your application requires *manual* fail over, you can configure your secondaries as *priority 0* (page 570). Priority 0 secondaries require manual action for a failover. This may be practical for a small replica set, but large deployments should fail over automatically.

See also:

replica set rollbacks (page 587).

Sharding Considerations

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

  To shard large amounts of data, create a new empty sharded collection, and ingest the data from the source collection using an application level import operation.
- Unique indexes are not enforced across shards except for the shard key itself. See *Enforce Unique Keys for Sharded Collections* (page 749).
- Consider *pre-splitting* (page 704) a sharded collection before a massive bulk import.

Analyze Performance

As you develop and operate applications with MongoDB, you may want to analyze the performance of the database as the application. Consider the following as you begin to investigate the performance of MongoDB.

Overview Degraded performance in MongoDB is typically a function of the relationship between the quantity of data stored in the database, the amount of system RAM, the number of connections to the database, and the amount of time the database spends in a locked state.

In some cases performance issues may be transient and related to traffic load, data access patterns, or the availability of hardware on the host system for virtualized environments. Some users also experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. In other situations, performance issues may indicate that the database may be operating at capacity and that it is time to add additional capacity to the database.

The following are some causes of degraded performance in MongoDB.

Locks MongoDB uses a locking system to ensure data set consistency. However, if certain operations are long-running, or a queue forms, performance will slow as requests and operations wait for the lock. Lock-related slowdowns can be intermittent. To see if the lock has been affecting your performance, look to the data in the *globalLock* section of the `serverStatus` output. If `globalLock.currentQueue.total` is consistently high, then there is a chance that a large number of requests are waiting for a lock. This indicates a possible concurrency issue that may be affecting performance.

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

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 229) and disk reads.

Memory Use MongoDB uses memory mapped files to store data. Given a data set of sufficient size, the MongoDB process will allocate all available memory on the system for its use. While this is part of the design, and affords MongoDB superior performance, the memory mapped files make it difficult to determine if the amount of RAM is sufficient for the data set.

The *memory usage statuses* metrics of the `serverStatus` output can provide insight into MongoDB's memory use. Check the resident memory use (i.e. `mem.resident`): if this exceeds the amount of system memory *and* there is a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

You should also check the amount of mapped memory (i.e. `mem.mapped`.) If this value is greater than the amount of system memory, some operations will require disk access *page faults* to read data from virtual memory and negatively affect performance.

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

Page faults triggered by MongoDB are reported as the total number of page faults in one second. To check for page faults, see the `extra_info.page_faults` value in the `serverStatus` output.

MongoDB on Windows counts both hard and soft page faults.

The MongoDB page fault counter may increase dramatically in moments of poor performance and may correlate with limited physical memory environments. Page faults also can increase while accessing much larger data sets, for example, scanning an entire collection. Limited and sporadic MongoDB page faults do not necessarily indicate a problem or a need to tune the database.

A single page fault completes quickly and is not problematic. However, in aggregate, large volumes of page faults typically indicate that MongoDB is reading too much data from disk. In many situations, MongoDB's read locks will "yield" after a page fault to allow other processes to read and avoid blocking while waiting for the next page to read into memory. This approach improves concurrency, and also improves overall throughput in high volume systems.

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

See *What are page faults?* (page 793) for more information.

Number of Connections In some cases, the number of connections between the application layer (i.e. clients) and the database can overwhelm the ability of the server to handle requests. This can produce performance irregularities. The following fields in the `serverStatus` document can provide insight:

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

If requests are high because there are numerous concurrent application requests, the database may have trouble keeping up with demand. If this is the case, then you will need to increase the capacity of your deployment. For read-heavy applications increase the size of your *replica set* and distribute read operations to *secondary* members. For write heavy applications, deploy *sharding* and add one or more *shards* to a *sharded cluster* to distribute load among `mongod` instances.

Spikes in the number of connections can also be the result of application or driver errors. All of the officially supported MongoDB drivers implement connection pooling, which allows clients to use and reuse connections more efficiently.

Extremely high numbers of connections, particularly without corresponding workload is often indicative of a driver or other configuration error.

Unless constrained by system-wide limits MongoDB has no limit on incoming connections. You can modify system limits using the `ulimit` command, or by editing your system's `/etc/sysctl` file. See *UNIX ulimit Settings* (page 300) for more information.

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

The following profiling levels are available:

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

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

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

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

See

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

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

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

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

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

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

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

See also:

Optimization Strategies for MongoDB (page 223) addresses strategies that may improve the performance of your database queries and operations.

Additional Resources

- MongoDB Ops Optimization Consulting Package[73]

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

5.2 Administration Tutorials

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

Configuration, Maintenance, and Analysis **(page 231)** Describes routine management operations, including configuration and performance analysis.

Manage mongod Processes **(page 236)** Start, configure, and manage running `mongod` process.

Rotate Log Files **(page 243)** Archive the current log files and start new ones.

Continue reading from *Configuration, Maintenance, and Analysis* (page 231) for additional tutorials of fundamental MongoDB maintenance procedures.

Backup and Recovery **(page 256)** Outlines procedures for data backup and restoration with `mongod` instances and deployments.

Backup and Restore with Filesystem Snapshots **(page 256)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

Backup and Restore Sharded Clusters **(page 265)** Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown **(page 274)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

Continue reading from *Backup and Recovery* (page 256) for additional tutorials of MongoDB backup and recovery procedures.

MongoDB Scripting **(page 277)** An introduction to the scripting capabilities of the `mongo` shell and the scripting capabilities embedded in MongoDB instances.

MongoDB Tutorials **(page 296)** A complete list of tutorials in the MongoDB Manual that address MongoDB operation and use.

5.2.1 Configuration, Maintenance, and Analysis

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

Disable Transparent Huge Pages (THP) **(page 232)** Describes Transparent Huge Pages (THP) and provides detailed instructions on disabling them.

Use Database Commands **(page 234)** The process for running database commands that provide basic database operations.

Manage mongod Processes **(page 236)** Start, configure, and manage running `mongod` process.

Terminate Running Operations **(page 238)** Stop in progress MongoDB client operations using `db.killOp()` and `maxTimeMS()`.

Analyze Performance of Database Operations **(page 239)** Collect data that introspects the performance of query and update operations on a `mongod` instance.

Rotate Log Files **(page 243)** Archive the current log files and start new ones.

Manage Journaling **(page 245)** Describes the procedures for configuring and managing MongoDB's journaling system which allows MongoDB to provide crash resiliency and durability.

Store a JavaScript Function on the Server **(page 247)** Describes how to store JavaScript functions on a MongoDB server.

Upgrade to the Latest Revision of MongoDB **(page 247)** Introduces the basic process for upgrading a MongoDB deployment between different minor release versions.

Monitor MongoDB With SNMP on Linux **(page 250)** The SNMP extension, available in MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

Monitor MongoDB Windows with SNMP **(page 252)** The SNMP extension, available in the Windows build of MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

Troubleshoot SNMP **(page 254)** Outlines common errors and diagnostic processes useful for deploying MongoDB Enterprise with SNMP support.

Disable Transparent Huge Pages (THP)

On this page

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

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

Init Script

Important: If you are using `tuned` or `ktune` (for example, if you are running Red Hat or CentOS 6+), you must additionally configure them so that THP is not re-enabled. See *Using tuned and ktune* (page 233).

Step 1: Create the `init.d` script. Create the following file at `/etc/init.d/disable-transparent-hugepages`:

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

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

```
      return 0
    fi

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

    unset thp_path
    ;;
esac
```

Step 2: Make it executable. Run the following command to ensure that the init script can be used:

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

Step 3: Configure your operating system to run it on boot. Use the appropriate command to configure the new init script on your Linux distribution.

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

Step 4: Override tuned and ktune, if applicable If you are using `tuned` or `ktune` (for example, if you are running Red Hat or CentOS 6+) you must now configure them to preserve the above settings.

Using `tuned` and `ktune`

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

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

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

Red Hat/CentOS 6

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

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

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

```
set_transparent_hugepages never
```

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

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

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

Red Hat/CentOS 7

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

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

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

```
[main]
include=virtual-guest

[vm]
transparent_hugepages=never
```

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

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

Test Your Changes

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

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

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

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

For both files, the correct output resembles:

```
always madvise [never]
```

Use Database Commands

On this page

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

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

Database Command Form

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

```
{ isMaster: 1 }
```

Issue Commands

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

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

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

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

You can use the maxTimeMS option to specify a time limit for the execution of a command, see *Terminate a Command* (page 239) for more information on operation termination.

admin Database Commands

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

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

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

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

Command Responses

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

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

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

Manage `mongod` Processes

On this page

MongoDB runs as a standard program. You can start MongoDB from a command line by issuing the `mongod` command and specifying options. For a list of options, see the `mongod` reference. MongoDB can also run as a Windows service. For details, see *Configure a Windows Service for MongoDB* (page 25). To install MongoDB, see *Install MongoDB* (page 5).

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

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

Start `mongod` Processes

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

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

```
mongod
```

Specify a Data Directory If you want `mongod` to store data files at a path *other than* `/data/db` you can specify a `dbPath`. The `dbPath` must exist before you start `mongod`. If it does not exist, create the directory and the permissions so that `mongod` can read and write data to this path. For more information on permissions, see the *security operations documentation* (page 431).

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

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

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

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

```
mongod --port 12345
```

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

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

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

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

Additional Configuration Options For an overview of common configurations and common configuration deployments. configurations for common use cases, see *Run-time Database Configuration* (page 203).

Stop `mongod` Processes

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

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

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

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

Calling the same method from a control script accomplishes the same result.

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

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

```
mongod --shutdown
```

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

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

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

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

Stop a Replica Set

Procedure If the `mongod` is the *primary* in a *replica set*, the shutdown process for these `mongod` instances has the following steps:

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

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

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

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

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

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

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

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

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

Terminate Running Operations

On this page

Overview

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

Available Procedures

`maxTimeMS` New in version 2.6.

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

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

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

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

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

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

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

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

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

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

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

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

Related

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

Analyze Performance of Database Operations

On this page

The database profiler collects fine grained data about MongoDB write operations, cursors, database commands on a running `mongod` instance. You can enable profiling on a per-database or per-instance basis. The *profiling level* (page 240) is also configurable when enabling profiling.

The database profiler writes all the data it collects to the `system.profile` (page 304) collection, which is a *capped collection* (page 219). See *Database Profiler Output* (page 305) for overview of the data in the `system.profile` (page 304) 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 305)
- `Profile Command`

- db.currentOp()

Profiling Levels

The following profiling levels are available:

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

  You can modify the threshold for "slow" operations with the slowOpThresholdMs runtime option or the setParameter command. See the *Specify the Threshold for Slow Operations* (page 240) section for more information.
- 2 - collects profiling data for all database operations.

Enable Database Profiling and Set the Profiling Level

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

When you enable profiling, you also set the *profiling level* (page 240). The profiler records data in the system.profile (page 304) collection. MongoDB creates the system.profile (page 304) collection in a database after you enable profiling for that database.

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

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

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

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

To verify the new setting, see the *Check Profiling Level* (page 241) section.

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

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

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

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

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

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

Check Profiling Level To view the *profiling level* (page 240), issue the following from the `mongo` shell:

```
db.getProfilingStatus()
```

The shell returns a document similar to the following:

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

The `was` field indicates the current level of profiling.

The `slowms` field indicates how long an operation must exist in milliseconds for an operation to pass the "slow" threshold. MongoDB will log operations that take longer than the threshold if the profiling level is `1`. This document returns the profiling level in the `was` field. For an explanation of profiling levels, see *Profiling Levels* (page 240).

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

```
db.getProfilingLevel()
```

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

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

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

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

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

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

See also:

`mode` and `slowOpThresholdMs`.

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

View Profiler Data

The database profiler logs information about database operations in the `system.profile` (page 304) collection.

To view profiling information, query the `system.profile` (page 304) collection. You can use `$comment` to add data to the query document to make it easier to analyze data from the profiler. To view example queries, see *Profiler Overhead* (page 242).

For an explanation of the output data, see *Database Profiler Output* (page 305).

Example Profiler Data Queries This section displays example queries to the `system.profile` (page 304) collection. For an explanation of the query output, see *Database Profiler Output* (page 305).

To return the most recent 10 log entries in the `system.profile` (page 304) collection, run a query similar to the following:

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

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

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

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

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

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

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

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

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

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

```
db.system.profile.find(
                        {
                          ts : {
                                 $gt : new ISODate("2011-07-12T03:00:00Z") ,
                                 $lt : new ISODate("2011-07-12T03:40:00Z")
                                }
                        },
                        { user : 0 }
                       ).sort( { millis : -1 } )
```

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

```
show profile
```

Profiler Overhead

When enabled, profiling has a minor effect on performance. The `system.profile` (page 304) collection is a *capped collection* with a default size of 1 megabyte. A collection of this size can typically store several thousand profile documents, but some application may use more or less profiling data per operation.

Change Size of `system.profile` Collection on the Primary To change the size of the `system.profile` (page 304) collection, you must:

1. Disable profiling.
2. Drop the `system.profile` (page 304) collection.
3. Create a new `system.profile` (page 304) collection.
4. Re-enable profiling.

For example, to create a new `system.profile` (page 304) collections that's `4000000` bytes, use the following sequence of operations in the `mongo` shell:

```
db.setProfilingLevel(0)

db.system.profile.drop()

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

db.setProfilingLevel(1)
```

Change Size of `system.profile` Collection on a Secondary To change the size of the `system.profile` (page 304) collection on a *secondary*, you must stop the secondary, run it as a standalone, and then perform the steps above. When done, restart the standalone as a member of the replica set. For more information, see *Perform Maintenance on Replica Set Members* (page 636).

Rotate Log Files

On this page

Overview

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

MongoDB's standard approach to log rotation only rotates logs in response to the `logRotate` command, or when the `mongod` or `mongos` process receives a `SIGUSR1` signal from the operating system.

Alternately, you may configure mongod to send log data to `syslog`. In this case, you can take advantage of alternate logrotation tools.

See also:

For information on logging, see the *Process Logging* (page 198) section.

Log Rotation With MongoDB

The following steps create and rotate a log file:

1. Start a `mongod` with verbose logging, with appending enabled, and with the following log file:

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

2. In a separate terminal, list the matching files:

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

   For results, you get:

   ```
   server1.log
   ```

3. Rotate the log file using *one* of the following methods.

   - From the `mongo` shell, issue the `logRotate` command from the `admin` database:

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

     This is the only available method to rotate log files on Windows systems.

   - For Linux systems, rotate logs for a single process by issuing the following command:

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

4. List the matching files again:

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

   For results you get something similar to the following. The timestamps will be different.

   ```
   server1.log  server1.log.2011-11-24T23-30-00
   ```

   The example results indicate a log rotation performed at exactly `11:30 pm` on `November 24th, 2011 UTC`, which is the local time offset by the local time zone. The original log file is the one with the timestamp. The new log is `server1.log` file.

   If you issue a second `logRotate` command an hour later, then an additional file would appear when listing matching files, as in the following example:

   ```
   server1.log  server1.log.2011-11-24T23-30-00  server1.log.2011-11-25T00-30-00
   ```

   This operation does not modify the `server1.log.2011-11-24T23-30-00` file created earlier, while `server1.log.2011-11-25T00-30-00` is the previous `server1.log` file, renamed. `server1.log` is a new, empty file that receives all new log output.

Syslog Log Rotation

New in version 2.2.

To configure mongod to send log data to syslog rather than writing log data to a file, use the following procedure.

1. Start a `mongod` with the `syslogFacility` option.
2. Store and rotate the log output using your system's default log rotation mechanism.

Important: You cannot use `syslogFacility` with `systemLog.path`.

Manage Journaling

On this page

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 77) durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal. If MongoDB should terminate or encounter an error before it can write the changes from the journal to the data files, MongoDB can re-apply the write operation and maintain a consistent state.

Without a journal, if `mongod` exits unexpectedly, you must assume your data is in an inconsistent state, and you must run either *repair* (page 274) or, preferably, *resync* (page 640) from a clean member of the replica set.

With journaling enabled, if `mongod` stops unexpectedly, the program can recover everything written to the journal, and the data remains in a consistent state. By default, the greatest extent of lost writes, i.e., those not made to the journal, are those made in the last 100 milliseconds. See `commitIntervalMs` 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 310).

Important: Changed in version 2.0: For 64-bit builds of `mongod`, journaling is enabled by default. For other platforms, see `storage.journal.enabled`.

Procedures

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

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

If no journal files exist, when `mongod` starts, it must preallocate new journal files. During this operation, the `mongod` is not listening for connections until preallocation completes: for some systems this may take a several minutes. During this period your applications and the `mongo` shell are not available.

Disable Journaling

Warning: Do not disable journaling on production systems. If your `mongod` instance stops without shutti down cleanly unexpectedly for any reason, (e.g. power failure) and you are not running with journaling, then y must recover from an unaffected *replica set* member or backup, as described in *repair* (page 274).

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

Get Commit Acknowledgment You can get commit acknowledgment with the *Write Concern* (page 82) and the `j` option. For details, see *Write Concern Reference* (page 135).

Avoid Preallocation Lag To avoid *preallocation lag* (page 309), you can preallocate files in the journal directory by copying them from another instance of `mongod`.

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

Example

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

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

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

   ```
   mkdir ~/tmpDbpath
   ```

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

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

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

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

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

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

5. Start the new `mongod` instance:

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

Monitor Journal Status Use the following commands and methods to monitor journal status:

- `serverStatus`

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

- `journalLatencyTest`

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

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

Change the Group Commit Interval Changed in version 2.0.

You can set the group commit interval using the *`--journalCommitInterval`* command line option. The allowed range is `2` to `300` milliseconds.

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

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

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

Store a JavaScript Function on the Server

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

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

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

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

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

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

Consider the following example from the `mongo` shell that first saves a function named `echoFunction` to the `system.js` collection and calls the function using `db.eval()` method:

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

db.eval( "echoFunction( 'test' )" )
```

See http://github.com/mongodb/mongo/tree/master/jstests/core/storefunc.js for a full example.

New in version 2.1: In the `mongo` shell, you can use `db.loadServerScripts()` to load all the scripts saved in the `system.js` collection for the current database. Once loaded, you can invoke the functions directly in the shell, as in the following example:

```
db.loadServerScripts();

echoFunction(3);

myAddFunction(3, 5);
```

Upgrade to the Latest Revision of MongoDB

On this page

Revisions provide security patches, bug fixes, and new or changed features that do not contain any backward breaking changes. Always upgrade to the latest revision in your release series. The third number in the *MongoDB version number* (page 908) indicates the revision.

Before Upgrading

- Ensure you have an up-to-date backup of your data set. See *MongoDB Backup Methods* (page 192).
- Consult the following documents for any special considerations or compatibility issues specific to your MongoDB release:
  - The release notes, located at *Release Notes* (page 805).
  - The documentation for your driver. See Drivers[74] page for more information.
- If your installation includes *replica sets*, plan the upgrade during a predefined maintenance window.
- Before you upgrade a production environment, use the procedures in this document to upgrade a *staging* environment that reproduces your production environment, to ensure that your production configuration is compatible with all changes.

Upgrade Procedure

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

Upgrade each `mongod` and `mongos` binary separately, using the procedure described here. When upgrading a binary, use the procedure *Upgrade a MongoDB Instance* (page 248).

Follow this upgrade procedure:

1. For deployments that use authentication, first upgrade all of your MongoDB `drivers`. To upgrade, see the documentation for your driver.
2. Upgrade sharded clusters, as described in *Upgrade Sharded Clusters* (page 249).
3. Upgrade any standalone instances. See *Upgrade a MongoDB Instance* (page 248).
4. Upgrade any replica sets that are not part of a sharded cluster, as described in *Upgrade Replica Sets* (page 250).

Upgrade a MongoDB Instance

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

[74] https://docs.mongodb.org/ecosystem/drivers

- Upgrade the instance using the operating system's package management tool and the official MongoDB packages. This is the preferred approach. See *Install MongoDB* (page 5).
- Upgrade the instance by replacing the existing binaries with new binaries. See *Replace the Existing Binaries* (page 249).

Replace the Existing Binaries

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

This section describes how to upgrade MongoDB by replacing the existing binaries. The preferred approach to an upgrade is to use the operating system's package management tool and the official MongoDB packages, as described in *Install MongoDB* (page 5).

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

1. Download the binaries for the latest MongoDB revision from the MongoDB Download Page[75] and store the binaries in a temporary location. The binaries download as compressed files that uncompress to the directory structure used by the MongoDB installation.
2. Shutdown the instance.
3. Replace the existing MongoDB binaries with the downloaded binaries.
4. Restart the instance.

Upgrade Sharded Clusters

To upgrade a sharded cluster:

1. Disable the cluster's balancer, as described in *Disable the Balancer* (page 732).
2. Upgrade each `mongos` instance by following the instructions below in *Upgrade a MongoDB Instance* (page 248). You can upgrade the `mongos` instances in any order.
3. Upgrade each `mongod` *config server* (page 684) 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 248)

   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 250).

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

- If a shard is a standalone instance, upgrade the shard using the procedure below titled *Upgrade a MongoDB Instance* (page 248).

5. Re-enable the balancer, as described in *Enable the Balancer* (page 733).

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` binary by following the instructions below in *Upgrade a MongoDB Instance* (page 248).
2. After upgrading a secondary, wait for the secondary to recover to the `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.

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

Upgrade the Primary

1. Step down the primary to initiate the normal *failover* (page 583) procedure. Using one of the following:
   - The `rs.stepDown()` helper in the `mongo` shell.
   - The `replSetStepDown` database command.

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

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

   ---

2. Once the primary has stepped down, call the `rs.status()` method from the `mongo` shell until you see that another member has assumed the `PRIMARY` state.
3. Shut down the original primary and upgrade its instance by following the instructions below in *Upgrade a MongoDB Instance* (page 248).

Additional Resources

- MongoDB Major Version Upgrade Consulting Package[76]

Monitor MongoDB With SNMP on Linux

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

On this page

New in version 2.2.

Enterprise Feature

SNMP is only available in MongoDB Enterprise[77].

Overview

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

See also:

Troubleshoot SNMP (page 254) and *Monitor MongoDB Windows with SNMP* (page 252) for complete instructions on using MongoDB with SNMP on Windows systems.

Considerations

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

Configuration Files

Changed in version 2.6.

MongoDB Enterprise contains the following configuration files to support SNMP:

- `MONGOD-MIB.txt`:

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

- `mongod.conf.subagent`:

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

- `mongod.conf.master`:

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

Procedure

Step 1: Copy configuration files. Use the following sequence of commands to move the SNMP configuration files to the SNMP service configuration directory.

[77] http://www.mongodb.com/products/mongodb-enterprise

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

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

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

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

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

Step 2: Start MongoDB. Start `mongod` with the `snmp-subagent` to send data to the SNMP master.

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

Step 3: Confirm SNMP data retrieval. Use `snmpwalk` to collect data from `mongod`:

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

Install the net-snmp[78] package to access the `snmpwalk` client. `net-snmp` provides the `snmpwalk` SNMP client.

```
snmpwalk -m /usr/share/snmp/mibs/MONGOD-MIB.txt -v 2c -c mongodb 127.0.0.1:<port> 1.3.6.1.4.1.34601
```

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

Optional: Run MongoDB as SNMP Master

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

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

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

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

Monitor MongoDB Windows with SNMP

On this page

[78] http://www.net-snmp.org/

New in version 2.6.

Enterprise Feature

SNMP is only available in MongoDB Enterprise[79].

Overview

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

See also:

Monitor MongoDB With SNMP on Linux (page 250) and *Troubleshoot SNMP* (page 254) for more information.

Considerations

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

Configuration Files

Changed in version 2.6.

MongoDB Enterprise contains the following configuration files to support SNMP:

- `MONGOD-MIB.txt`:

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

- `mongod.conf.subagent`:

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

- `mongod.conf.master`:

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

Procedure

Step 1: Copy configuration files. Use the following sequence of commands to move the SNMP configuration files to the SNMP service configuration directory.

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

```
md C:\snmp\etc\config
copy MONGOD-MIB.txt C:\snmp\etc\config\MONGOD-MIB.txt
copy mongod.conf.subagent C:\snmp\etc\config\mongod.conf
```

[79] http://www.mongodb.com/products/mongodb-enterprise

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

Edit the configuration file to ensure that the communication between the agent (i.e. `snmpd` or the master) and subagent (i.e. MongoDB) uses TCP.

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

Step 2: Start MongoDB. Start `mongod.exe` with the `snmp-subagent` to send data to the SNMP master.

```
mongod.exe --snmp-subagent
```

Step 3: Confirm SNMP data retrieval. Use `snmpwalk` to collect data from `mongod.exe`:

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

Install the net-snmp[80] package to access the `snmpwalk` client. `net-snmp` provides the `snmpwalk` SNMP client.

```
snmpwalk -m C:\snmp\etc\config\MONGOD-MIB.txt -v 2c -c mongodb 127.0.0.1:<port> 1.3.6.1.4.1.34601
```

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

Optional: Run MongoDB as SNMP Master

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

```
copy mongod.conf.master C:\snmp\etc\config\mongod.conf
```

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

```
mongod.exe --snmp-master
```

Troubleshoot SNMP

On this page

New in version 2.6.

Enterprise Feature

SNMP is only available in MongoDB Enterprise.

[80] http://www.net-snmp.org/

Overview

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This document identifies common problems you may encounter when deploying MongoDB Enterprise with SNMP as well as possible solutions for these issues.

See *Monitor MongoDB With SNMP on Linux* (page 250) and *Monitor MongoDB Windows with SNMP* (page 252) for complete installation instructions.

Issues

Failed to Connect The following in the `mongod` logfile:

```
Warning: Failed to connect to the agentx master agent
```

AgentX is the SNMP agent extensibility protocol defined in Internet RFC 2741[81]. It explains how to define additional data to monitor over SNMP. When MongoDB fails to connect to the agentx master agent, use the following procedure to ensure that the SNMP subagent can connect properly to the SNMP master.

1. Make sure the master agent is running.
2. Compare the SNMP master's configuration file with the subagent configuration file. Ensure that the agentx socket definition is the same between the two.
3. Check the SNMP configuration files to see if they specify using UNIX Domain Sockets. If so, confirm that the `mongod` has appropriate permissions to open a UNIX domain socket.

Error Parsing Command Line One of the following errors at the command line:

```
Error parsing command line: unknown option snmp-master
try 'mongod --help' for more information

Error parsing command line: unknown option snmp-subagent
try 'mongod --help' for more information
```

`mongod` binaries that are not part of the Enterprise Edition produce this error. *Install the Enterprise Edition* (page 27) and attempt to start `mongod` again.

Other MongoDB binaries, including `mongos` will produce this error if you attempt to star them with `snmp-master` or `snmp-subagent`. Only `mongod` supports SNMP.

Error Starting `SNMPAgent` The following line in the log file indicates that `mongod` cannot read the `mongod.conf` file:

```
[SNMPAgent] warning: error starting SNMPAgent as master err:1
```

If running on Linux, ensure `mongod.conf` exists in the `/etc/snmp` directory, and ensure that the `mongod` UNIX user has permission to read the `mongod.conf` file.

If running on Windows, ensure `mongod.conf` exists in `C:\snmp\etc\config`.

[81] http://www.ietf.org/rfc/rfc2741.txt

5.2.2 Backup and Recovery

The following tutorials describe backup and restoration for a `mongod` instance:

Backup and Restore with Filesystem Snapshots **(page 256)** An outline of procedures for creating MongoDB data set backups using system-level file snapshot tool, such as *LVM* or native storage appliance tools.

Restore a Replica Set from MongoDB Backups **(page 260)** Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or MongoDB Cloud Manager[82] Backup file.

Back Up and Restore with MongoDB Tools **(page 261)** The procedure for writing the contents of a database to a BSON (i.e. binary) dump file for backing up MongoDB databases.

Backup and Restore Sharded Clusters **(page 265)** Detailed procedures and considerations for backing up sharded clusters and single shards.

Recover Data after an Unexpected Shutdown **(page 274)** Recover data from MongoDB data files that were not properly closed or have an invalid state.

Backup and Restore with Filesystem Snapshots

On this page

This document describes a procedure for creating backups of MongoDB systems using system-level tools, such as *LVM* or storage appliance, as well as the corresponding restoration strategies.

These filesystem snapshots, or "block-level" backup methods use system level tools to create copies of the device that holds MongoDB's data files. These methods complete quickly and work reliably, but require more system configuration outside of MongoDB.

See also:

MongoDB Backup Methods (page 192) and *Back Up and Restore with MongoDB Tools* (page 261).

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 valid when the snapshot takes place. This means that all writes accepted by the database need to be fully written to disk: either to the *journal* or to data files.

  If all writes are not on disk when the backup occurs, the backup will not reflect these changes. 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

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

to disk before running the backup operation and must ensure that no writes occur during the entire backup procedure.

If you do use journaling, the journal **must** reside on the same volume as the data.

- Snapshots create an image of an entire disk image. Unless you need to back up your entire system, consider isolating your MongoDB data files, journal (if applicable), and configuration on one logical disk that doesn't contain any other data.

  Alternately, store all MongoDB data files on a dedicated device so that you can make backups without duplicating extraneous data.

- Ensure that you copy data from snapshots and onto other systems to ensure that data is safe from site failures.

- Although different snapshots methods provide different capability, the LVM method outlined below does not provide any capacity for capturing incremental backups.

Snapshots With Journaling If your `mongod` instance has journaling enabled, then you can use any kind of file system or volume/block level snapshot tool to create backups.

If you manage your own infrastructure on a Linux-based system, configure your system with *LVM* to provide your disk packages and provide snapshot capability. You can also use LVM-based setups *within* a cloud/virtualized environment.

Note: Running *LVM* provides additional flexibility and enables the possibility of using snapshots to back up MongoDB.

Snapshots with Amazon EBS in a RAID 10 Configuration If your deployment depends on Amazon's Elastic Block Storage (EBS) with RAID configured within your instance, it is impossible to get a consistent state across all disks using the platform's snapshot tool. As an alternative, you can do one of the following:

- Flush all writes to disk and create a write lock to ensure consistent state during the backup process.

  If you choose this option see *Create Backups on Instances that do not have Journaling Enabled* (page 259).

- 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 257).

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 `/dev/vg0/mdb-snap01`. The location and paths to your systems volume groups and devices may vary slightly depending on your operating system's *LVM* configuration.

The snapshot has a cap of at 100 megabytes, because of the parameter `--size 100M`. This size does not reflect the total amount of the data on the disk, but rather the quantity of differences between the current state of `/dev/vg0/mongodb` and the creation of the snapshot (i.e. `/dev/vg0/mdb-snap01`.)

> **Warning:** Ensure that you create snapshots with enough space to account for data growth, particularly for the period of time that it takes to copy data out of the system or to a temporary image.
> If your snapshot runs out of space, the snapshot image becomes unusable. Discard this logical volume and create another.

The snapshot will exist when the command returns. You can restore directly from the snapshot at any time or by creating a new logical volume and restoring from this snapshot to the alternate image.

While snapshots are great for creating high quality backups very quickly, they are not ideal as a format for storing backup data. Snapshots typically depend and reside on the same storage infrastructure as the original disk images. Therefore, it's crucial that you archive these snapshots and store them elsewhere.

Archive a Snapshot After creating a snapshot, mount the snapshot and copy the data to separate storage. Your system might try to compress the backup images as you move them offline. Alternatively, take a block level copy of the snapshot image, such as with the following procedure:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | gzip > mdb-snap01.gz
```

The above command sequence does the following:

- Ensures that the `/dev/vg0/mdb-snap01` device is not mounted. Never take a block level copy of a filesystem or filesystem snapshot that is mounted.
- Performs a block level copy of the entire snapshot image using the `dd` command and compresses the result in a gzipped file in the current working directory.

  > **Warning:** This command will create a large `gz` file in your current working directory. Make sure that you run this command in a file system that has enough free space.

Restore a Snapshot To restore a snapshot created with the above method, issue the following sequence of commands:

```
lvcreate --size 1G --name mdb-new vg0
gzip -d -c mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

The above sequence does the following:

- Creates a new logical volume named `mdb-new`, in the `/dev/vg0` volume group. The path to the new device will be `/dev/vg0/mdb-new`.

  > **Warning:** This volume will have a maximum size of 1 gigabyte. The original file system must have had a total size of 1 gigabyte or smaller, or else the restoration will fail.
  > Change `1G` to your desired volume size.

- Uncompresses and unarchives the `mdb-snap01.gz` into the `mdb-new` disk image.

- Mounts the `mdb-new` disk image to the `/srv/mongodb` directory. Modify the mount point to correspond to your MongoDB data file location, or other location as needed.

Note: The restored snapshot will have a stale `mongod.lock` file. If you do not remove this file from the snapshot, and MongoDB may assume that the stale lock file indicates an unclean shutdown. If you're running with `storage.journal.enabled` enabled, and you *do not* use `db.fsyncLock()`, you do not need to remove the `mongod.lock` file. If you use `db.fsyncLock()` you will need to remove the lock.

Restore Directly from a Snapshot To restore a backup without writing to a compressed `gz` file, use the following sequence of commands:

```
umount /dev/vg0/mdb-snap01
lvcreate --size 1G --name mdb-new vg0
dd if=/dev/vg0/mdb-snap01 of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Remote Backup Storage You can implement off-system backups using the *combined process* (page 259) and SSH.

This sequence is identical to procedures explained above, except that it archives and compresses the backup on a remote system using SSH.

Consider the following procedure:

```
umount /dev/vg0/mdb-snap01
dd if=/dev/vg0/mdb-snap01 | ssh username@example.com gzip > /opt/backup/mdb-snap01.gz
lvcreate --size 1G --name mdb-new vg0
ssh username@example.com gzip -d -c /opt/backup/mdb-snap01.gz | dd of=/dev/vg0/mdb-new
mount /dev/vg0/mdb-new /srv/mongodb
```

Create Backups on Instances that do not have Journaling Enabled

If your `mongod` instance does not run with journaling enabled, or if your journal is on a separate volume, obtaining a functional backup of a consistent state is more complicated. As described in this section, you must flush all writes to disk and lock the database to prevent writes during the backup process. If you have a *replica set* configuration, then for your backup use a *secondary* which is not receiving reads (i.e. *hidden member*).

Important: In the following procedure, you **must** issue the `db.fsyncLock()` and `db.fsyncUnlock()` operations on the same connection. The client that issues `db.fsyncLock()` is solely responsible for issuing a `db.fsyncUnlock()` operation and must be able to handle potential error conditions so that it can perform the `db.fsyncUnlock()` before terminating the connection.

Step 1: Flush writes to disk and lock the database to prevent further writes. To flush writes to disk and to "lock" the database, issue the `db.fsyncLock()` method in the `mongo` shell:

```
db.fsyncLock();
```

Step 2: Perform the backup operation described in *Create a Snapshot*.

Step 3: After the snapshot completes, unlock the database. To unlock the database after the snapshot has completed, use the following command in the `mongo` shell:

```
db.fsyncUnlock();
```

Changed in version 2.2: When used in combination with `fsync` or `db.fsyncLock()`, `mongod` will block reads, including those from `mongodump`, when queued write operation waits behind the `fsync` lock. Do not use `mongodump` with `db.fsyncLock()`.

Restore a Replica Set from MongoDB Backups

On this page

This procedure outlines the process for taking MongoDB data and restoring that data into a new *replica set*. Use this approach for seeding test deployments from production backups as well as part of disaster recovery.

You *cannot* restore a single data set to three new `mongod` instances and *then* create a replica set. In this situation MongoDB will force the secondaries to perform an initial sync. The procedures in this document describe the correct and efficient ways to deploy a replica set.

Restore Database into a Single Node Replica Set

Step 1: Obtain backup MongoDB Database files. The backup files may come from a *file system snapshot* (page 256). The MongoDB Cloud Manager[83] produces MongoDB database files for stored snapshots[84] and point in time snapshots[85].

You can also use `mongorestore` to restore database files using data created with `mongodump`. See *Back Up and Restore with MongoDB Tools* (page 261) for more information.

Step 2: Start a `mongod` using data files from the backup as the data path. Start a `mongod` for a new single-node replica set. Specify the path to the backup data files with `--dbpath` option and the replica set name with the `--replSet` option.

```
mongod --dbpath /data/db --replSet <replName>
```

Step 3: Connect a `mongo` shell to the `mongod` instance. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 4: Initiate the new replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

[83] https://cloud.mongodb.com/?jmp=docs
[84] https://docs.cloud.mongodb.com/tutorial/restore-from-snapshot/
[85] https://docs.cloud.mongodb.com/tutorial/restore-from-point-in-time-snapshot/

Add Members to the Replica Set

MongoDB provides two options for restoring secondary members of a replica set:

- Manually copy the database files to each data directory.
- Allow *initial sync* (page 598) to distribute data automatically.

The following sections outlines both approaches.

Note: If your database is large, initial sync can take a long time to complete. For large databases, it might be preferable to copy the database files onto each host.

Copy Database Files and Restart `mongod` Instance Use the following sequence of operations to "seed" additional members of the replica set with the restored data by copying MongoDB data files directly.

Step 1: Shut down the `mongod` instance that you restored. Use `--shutdown` or `db.shutdownServer()` to ensure a clean shut down.

Step 2: Copy the primary's data directory to each secondary. Copy the *primary's* data directory into the `dbPath` of the other members of the replica set. The `dbPath` is `/data/db` by default.

Step 3: Start the `mongod` instance that you restored.

Step 4: Add the secondaries to the replica set. In a `mongo` shell connected to the *primary*, add the *secondaries* to the replica set using `rs.add()`. See *Deploy a Replica Set* (page 607) for more information about deploying a replica set.

Update Secondaries using Initial Sync Use the following sequence of operations to "seed" additional members of the replica set with the restored data using the default *initial sync* operation.

Step 1: Ensure that the data directories on the prospective replica set members are empty.

Step 2: Add each prospective member to the replica set. When you add a member to the replica set, *Initial Sync* (page 598) copies the data from the *primary* to the new member.

Back Up and Restore with MongoDB Tools

On this page

This document describes the process for writing and restoring backups to files in binary format with the `mongodump` and `mongorestore` tools.

Use these tools for backups if other backup methods, such as the MongoDB Cloud Manager[86] or *file system snapshots* (page 256) are unavailable.

See also:

MongoDB Backup Methods (page 192), `mongodump`, and `mongorestore`.

Backup a Database with `mongodump`

`mongodump` does *not* dump the content of the `local` database.

To backup all the databases in a cluster via `mongodump`, you should have the `backup` (page 410) role. The `backup` (page 410) role provides all the needed privileges for backing up all database. The role confers no additional access, in keeping with the policy of *least privilege*.

To backup a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` (page 410) role.

To backup the `system.profile` collection in a database, you must have `read` access on certain system collections in the database. Several roles provide this access, including the `clusterAdmin` (page 407) and `dbAdmin` (page 406) roles.

Changed in version 2.6.

To backup users and *user-defined roles* (page 321) for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to backup a given database's users, you must have the `find` (page 419) *action* (page 418) on the `admin` database's `admin.system.users` (page 304) collection. The `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles both provide this privilege.

To backup the user-defined roles on a database, you must have the `find` (page 419) action on the `admin` database's `admin.system.roles` (page 304) collection. Both the `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles provide this privilege.

Basic `mongodump` Operations The `mongodump` utility can back up data by either:

- connecting to a running `mongod` or `mongos` instance, or
- accessing data files without an active instance.

The utility can create a backup for an entire server, database or collection, or can use a query to backup just part of a collection.

When you run `mongodump` without any arguments, the command connects to the MongoDB instance on the local system (e.g. `127.0.0.1` or `localhost`) on port `27017` and creates a database backup named `dump/` in the current directory.

To backup data from a `mongod` or `mongos` instance running on the same machine and on the default port of `27017`, use the following command:

```
mongodump
```

The data format used by `mongodump` from version 2.2 or later is *incompatible* with earlier versions of `mongod`. Do not use recent versions of `mongodump` to back up older data stores.

You can also specify the `--host` and `--port` of the MongoDB instance that the `mongodump` should connect to. For example:

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

```
mongodump --host mongodb.example.net --port 27017
```

`mongodump` will write *BSON* files that hold a copy of data accessible via the `mongod` listening on port `27017` of the `mongodb.example.net` host. See *Create Backups from Non-Local mongod Instances* (page 263) for more information.

To use `mongodump` without a running MongoDB instance, specify the *`--dbpath`* option to read directly from MongoDB data files. See *Create Backups Without a Running mongod Instance* (page 263) for details.

To specify a different output directory, you can use the *`--out or -o`* option:

```
mongodump --out /data/backup/
```

To limit the amount of data included in the database dump, you can specify *`--db`* and *`--collection`* as options to `mongodump`. For example:

```
mongodump --collection myCollection --db test
```

This operation creates a dump of the collection named `myCollection` from the database `test` in a `dump/` subdirectory of the current working directory.

`mongodump` overwrites output files if they exist in the backup data folder. Before running the `mongodump` command multiple times, either ensure that you no longer need the files in the output folder (the default is the `dump/` folder) or rename the folders or files.

Point in Time Operation Using Oplogs Use the *`--oplog`* option with `mongodump` to collect the *oplog* entries to build a point-in-time snapshot of a database within a replica set. With *`--oplog`*, `mongodump` copies all the data from the source database as well as all of the *oplog* entries from the beginning to the end of the backup procedure. This operation, in conjunction with *`mongorestore --oplogReplay`*, allows you to restore a backup that reflects the specific moment in time that corresponds to when `mongodump` completed creating the dump file.

Create Backups Without a Running `mongod` Instance If your MongoDB instance is not running, you can use the *`--dbpath`* option to specify the location to your MongoDB instance's database files. `mongodump` reads from the data files directly with this operation. This locks the data directory to prevent conflicting writes. The `mongod` process must *not* be running or attached to these data files when you run `mongodump` in this configuration. Consider the following example:

Given a MongoDB instance that contains the `customers`, `products`, and `suppliers` databases, the following `mongodump` operation backs up the databases using the *`--dbpath`* option, which specifies the location of the database files on the host:

```
mongodump --dbpath /data -o dataout
```

The *`--out or -o`* option allows you to specify the directory where `mongodump` will save the backup. `mongodump` creates a separate backup directory for each of the backed up databases: `dataout/customers`, `dataout/products`, and `dataout/suppliers`.

Create Backups from Non-Local `mongod` Instances The *`--host`* and *`--port`* options for `mongodump` allow you to connect to and backup from a remote host. Consider the following example:

```
mongodump --host mongodb1.example.net --port 3017 --username user --password pass --out /opt/backup/m
```

On any `mongodump` command you may, as above, specify username and password credentials to specify database authentication.

Restore a Database with `mongorestore`

Changed in version 2.6.

To restore users and *user-defined roles* (page 321) on a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to restore users to a given database, you must have the `insert` (page 419) *action* (page 418) on the `admin` database's `admin.system.users` (page 304) collection. The `restore` (page 410) role provides this privilege.

To restore user-defined roles to a database, you must have the `insert` (page 419) action on the `admin` database's `admin.system.roles` (page 304) collection. The `restore` (page 410) role provides this privilege.

Basic `mongorestore` Operations The `mongorestore` utility restores a binary backup created by `mongodump`. By default, `mongorestore` looks for a database backup in the `dump/` directory.

The `mongorestore` utility can restore data either by:

- connecting to a running `mongod` or `mongos` directly, or
- writing to a set of MongoDB data files without use of a running `mongod`.

`mongorestore` can restore either an entire database backup or a subset of the backup.

To use `mongorestore` to connect to an active `mongod` or `mongos`, use a command with the following prototype form:

```
mongorestore --port <port number> <path to the backup>
```

To use `mongorestore` to write to data files without using a running `mongod`, use a command with the following prototype form:

```
mongorestore --dbpath <database path> <path to the backup>
```

Consider the following example:

```
mongorestore dump-2013-10-25/
```

Here, `mongorestore` imports the database backup in the `dump-2013-10-25` directory to the `mongod` instance running on the localhost interface.

Restore Point in Time Oplog Backup If you created your database dump using the *`--oplog`* option to ensure a point-in-time snapshot, call `mongorestore` with the *`--oplogReplay`* option, as in the following example:

```
mongorestore --oplogReplay
```

You may also consider using the *`mongorestore --objcheck`* option to check the integrity of objects while inserting them into the database, or you may consider the *`mongorestore --drop`* option to drop each collection from the database before restoring from backups.

Restore a Subset of data from a Binary Database Dump `mongorestore` also includes the ability to a filter to all input before inserting it into the new database. Consider the following example:

```
mongorestore --filter '{"field": 1}'
```

Here, `mongorestore` only adds documents to the database from the dump located in the `dump/` folder *if* the documents have a field name `field` that holds a value of `1`. Enclose the filter in single quotes (e.g. `'`) to prevent the filter from interacting with your shell environment.

Restore Without a Running `mongod` `mongorestore` can write data to MongoDB data files without needing to connect to a `mongod` directly.

Example

Restore a Database Without a Running `mongod`

Given a set of backed up databases in the `/data/backup/` directory:

- `/data/backup/customers`,
- `/data/backup/products`, and
- `/data/backup/suppliers`

The following `mongorestore` command restores the `products` database. The command uses the *`--dbpath`* option to specify the path to the MongoDB data files:

```
mongorestore --dbpath /data/db --journal /data/backup/products
```

The `mongorestore` imports the database backup in the `/data/backup/products` directory to the `mongod` instance that runs on the localhost interface. The `mongorestore` operation imports the backup even if the `mongod` is not running.

The *`--journal`* option ensures that `mongorestore` records all operation in the durability *journal*. The journal prevents data file corruption if anything (e.g. power failure, disk failure, etc.) interrupts the restore operation.

Restore Backups to Non-Local `mongod` Instances By default, `mongorestore` connects to a MongoDB instance running on the localhost interface (e.g. `127.0.0.1`) and on the default port (`27017`). If you want to restore to a different host or port, use the *`--host`* and *`--port`* options.

Consider the following example:

```
mongorestore --host mongodb1.example.net --port 3017 --username user --password pass /opt/backup/mong
```

As above, you may specify username and password connections if your `mongod` requires authentication.

Additional Resources

- Backup and its Role in Disaster Recovery White Paper[87]
- Cloud Backup through MongoDB Cloud Manager[88]
- Blog Post: Backup vs. Replication, Why you Need Both[89]

Backup and Restore Sharded Clusters

The following tutorials describe backup and restoration for sharded clusters:

Backup a Small Sharded Cluster with mongodump **(page 266)** If your *sharded cluster* holds a small data set, you can use `mongodump` to capture the entire backup in a reasonable amount of time.

Backup a Sharded Cluster with Filesystem Snapshots **(page 267)** Use file system snapshots back up each component in the sharded cluster individually. The procedure involves stopping the cluster balancer. If your system configuration allows file system backups, this might be more efficient than using MongoDB tools.

[87] https://www.mongodb.com/lp/white-paper/backup-disaster-recovery?jmp=docs
[88] https://cloud.mongodb.com/?jmp=docs
[89] http://www.mongodb.com/blog/post/backup-vs-replication-why-do-you-need-both?jmp=docs

Backup a Sharded Cluster with Database Dumps **(page 269)** Create backups using `mongodump` to back up each component in the cluster individually.

Schedule Backup Window for Sharded Clusters **(page 271)** Limit the operation of the cluster balancer to provide a window for regular backup operations.

Restore a Single Shard **(page 271)** An outline of the procedure and consideration for restoring a single shard from a backup.

Restore a Sharded Cluster **(page 272)** An outline of the procedure and consideration for restoring an *entire* sharded cluster from backup.

Backup a Small Sharded Cluster with `mongodump`

On this page

Overview If your *sharded cluster* holds a small data set, you can connect to a `mongos` using `mongodump`. You can create backups of your MongoDB cluster, if your backup infrastructure can capture the entire backup in a reasonable amount of time and if you have a storage system that can hold the complete MongoDB data set.

See *MongoDB Backup Methods* (page 192) and *Backup and Restore Sharded Clusters* (page 265) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Important: By default `mongodump` issue its queries to the non-primary nodes.

To backup all the databases in a cluster via `mongodump`, you should have the `backup` (page 410) role. The `backup` (page 410) role provides all the needed privileges for backing up all database. The role confers no additional access, in keeping with the policy of *least privilege*.

To backup a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` (page 410) role.

To backup the `system.profile` collection in a database, you must have `read` access on certain system collections in the database. Several roles provide this access, including the `clusterAdmin` (page 407) and `dbAdmin` (page 406) roles.

Changed in version 2.6.

To backup users and *user-defined roles* (page 321) for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to backup a given database's users, you must have the `find` (page 419) *action* (page 418) on the `admin` database's `admin.system.users` (page 304) collection. The `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles both provide this privilege.

To backup the user-defined roles on a database, you must have the `find` (page 419) action on the `admin` database's `admin.system.roles` (page 304) collection. Both the `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles provide this privilege.

Considerations If you use `mongodump` without specifying a database or collection, `mongodump` will capture collection data *and* the cluster meta-data from the *config servers* (page 684).

You cannot use the `--oplog` option for `mongodump` when capturing data from `mongos`. As a result, if you need to capture a backup that reflects a single moment in time, you must stop all writes to the cluster for the duration of the backup operation.

Procedure

Capture Data You can perform a backup of a *sharded cluster* by connecting `mongodump` to a `mongos`. Use the following operation at your system's prompt:

```
mongodump --host mongos3.example.net --port 27017
```

`mongodump` will write *BSON* files that hold a copy of data stored in the *sharded cluster* accessible via the `mongos` listening on port `27017` of the `mongos3.example.net` host.

Restore Data Backups created with `mongodump` do not reflect the chunks or the distribution of data in the sharded collection or collections. Like all `mongodump` output, these backups contain separate directories for each database and *BSON* files for each collection in that database.

You can restore `mongodump` output to any MongoDB instance, including a standalone, a *replica set*, or a new *sharded cluster*. When restoring data to sharded cluster, you must deploy and configure sharding before restoring data from the backup. See *Deploy a Sharded Cluster* (page 705) for more information.

Backup a Sharded Cluster with Filesystem Snapshots

On this page

Overview This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses file system snapshots to capture a copy of the `mongod` instance. An alternate procedure uses `mongodump` to create binary database dumps when file-system snapshots are not available. See *Backup a Sharded Cluster with Database Dumps* (page 269) for the alternate procedure.

See *MongoDB Backup Methods* (page 192) and *Backup and Restore Sharded Clusters* (page 265) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Considerations

Balancing It is *essential* that you stop the balancer before capturing a backup.

If the balancer is active while you capture backups, the backup artifacts may be incomplete and/or have duplicate data, as *chunks* may migrate while recording backups.

Precision In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using a file-system snapshot tool. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

Consistency If the journal and data files are on the same logical volume, you can use a single point-in-time snapshot to capture a valid copy of the data.

If the journal and data files are on different file systems, you must use `db.fsyncLock()` and `db.fsyncUnLock()` to capture a valid copy of your data.

Procedure

Step 1: Disable the balancer. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell.

Consider the following example:

```
use config
sh.stopBalancer()
```

For more information, see the *Disable the Balancer* (page 732) procedure.

Step 2: If necessary, lock one secondary member of each replica set in each shard. If your `mongod` does not have journaling enabled *or* your journal and data files are on different volumes, you **must** lock your `mongod` before capturing a back up.

If your `mongod` has journaling enabled and your journal and data files are on the same volume, you may skip this step.

If you need to lock the `monogd`, attempt to lock one secondary member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time.

To lock a secondary, connect through the `mongo` shell to the secondary member's `mongod` instance and issue the `db.fsyncLock()` method.

Step 3: Back up one of the config servers. Backing up a *config server* (page 684) backs up the sharded cluster's metadata. You need back up only one config server, as they all hold the same data. Do one of the following to back up one of the config servers:

Create a file-system snapshot of the config server. Do this **only if** the config server has *journaling* enabled. Use the procedure in *Backup and Restore with Filesystem Snapshots* (page 256). **Never** use `db.fsyncLock()` on config databases.

Create a database dump to backup the config server. Issue `mongodump` against one of the config `mongod` instances. If you are running MongoDB 2.4 or later with the *`--configsvr`* option, then include the *`--oplog`* option to ensure that the dump includes a partial oplog containing operations from the duration of the mongodump operation. For example:

```
mongodump --oplog
```

Step 4: Back up the replica set members of the shards that you locked. You may back up the shards in parallel. For each shard, create a snapshot. Use the procedure in *Backup and Restore with Filesystem Snapshots* (page 256).

Step 5: Unlock locked replica set members. If you locked any `mongod` instances to capture the backup, unlock them now.

Unlock all locked replica set members of each shard using the `db.fsyncUnlock()` method in the `mongo` shell.

Step 6: Enable the balancer. Re-enable the balancer with the `sh.setBalancerState()` method. Use the following command sequence when connected to the `mongos` with the `mongo` shell:

```
use config
sh.setBalancerState(true)
```

Backup a Sharded Cluster with Database Dumps

On this page

Overview This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses `mongodump` to create dumps of the `mongod` instance. An alternate procedure uses file system snapshots to capture the backup data, and may be more efficient in some situations if your system configuration allows file system backups. See *Backup and Restore Sharded Clusters* (page 265) for more information.

See *MongoDB Backup Methods* (page 192) and *Backup and Restore Sharded Clusters* (page 265) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Prerequisites

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

To backup all the databases in a cluster via `mongodump`, you should have the `backup` (page 410) role. The `backup` (page 410) role provides all the needed privileges for backing up all database. The role confers no additional access, in keeping with the policy of *least privilege*.

To backup a given database, you must have `read` access on the database. Several roles provide this access, including the `backup` (page 410) role.

To backup the `system.profile` collection in a database, you must have `read` access on certain system collections in the database. Several roles provide this access, including the `clusterAdmin` (page 407) and `dbAdmin` (page 406) roles.

Changed in version 2.6.

To backup users and *user-defined roles* (page 321) for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to backup a given database's users, you must have the `find` (page 419) *action* (page 418) on the `admin` database's `admin.system.users` (page 304) collection. The `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles both provide this privilege.

To backup the user-defined roles on a database, you must have the `find` (page 419) action on the `admin` database's `admin.system.roles` (page 304) collection. Both the `backup` (page 410) and `userAdminAnyDatabase` (page 411) roles provide this privilege.

Consideration To create these backups of a sharded cluster, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using `mongodump` to capture the backup data. To capture a more exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, taking the backup from a single offline secondary member of the replica set that provides each shard can improve the quality of the backup while minimizing impact on the cluster.

Procedure

Step 1: Disable the balancer process. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell. For example:

```
use config
sh.setBalancerState(false)
```

For more information, see the *Disable the Balancer* (page 732) procedure.

> **Warning:** If you do not stop the balancer, the backup could have duplicate data or omit data as *chunks* migrate while recording backups.

Step 2: Lock replica set members. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` instances in as short of an interval as possible.

To lock or freeze a sharded cluster, you shut down 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 597) for more information.

Step 3: Backup one config server. Run `mongodump` against a config server `mongod` instance to back up the cluster's metadata. The config server `mongod` instance must be version 2.4 or later and must run with the `--configsvr` option. You only need to back up one config server.

Use `mongodump` with the `--oplog` option to backup one of the *config servers* (page 684).

```
mongodump --oplog
```

Step 4: Backup replica set members. Back up the replica set members of the shards that shut down using `mongodump` and specifying the `--dbpath` option. You may back up the shards in parallel. Consider the following invocation:

```
mongodump --journal --dbpath /data/db/ --out /data/backup/
```

You must run `mongodump` on the same system where the `mongod` ran. This operation will create a dump of all the data managed by the `mongod` instances that used the `dbPath /data/db/`. `mongodump` writes the output of this dump to the `/data/backup/` directory.

Step 5: Restart replica set members. Restart all stopped replica set members of each shard as normal and allow them to catch up with the state of the primary.

Step 6: Re-enable the balancer process. Re-enable the balancer with the `sh.setBalancerState()` method.

Use the following command sequence when connected to the `mongos` with the `mongo` shell:

```
use config
sh.setBalancerState(true)
```

Schedule Backup Window for Sharded Clusters

On this page

Overview In a *sharded cluster*, the balancer process is responsible for distributing sharded data around the cluster, so that each *shard* has roughly the same amount of data.

However, when creating backups from a sharded cluster it is important that you disable the balancer while taking backups to ensure that no chunk migrations affect the content of the backup captured by the backup procedure. Using the procedure outlined in the section *Disable the Balancer* (page 732) you can manually stop the balancer process temporarily. As an alternative you can use this procedure to define a balancing window so that the balancer is always disabled during your automated backup operation.

Procedure If you have an automated backup schedule, you can disable all balancing operations for a period of time. For instance, consider the following command:

```
use config
db.settings.update( { _id : "balancer" }, { $set : { activeWindow : { start : "6:00", stop : "23:00"
```

This operation configures the balancer to run between 6:00am and 11:00pm, server time. Schedule your backup operation to run *and complete* outside of this time. Ensure that the backup can complete outside the window when the balancer is running *and* that the balancer can effectively balance the collection among the shards in the window allotted to each.

Restore a Single Shard

On this page

Overview Restoring a single shard from backup with other unaffected shards requires a number of special considerations and practices. This document outlines the additional tasks you must perform when restoring a single shard.

Consider the following resources on backups in general as well as backup and restoration of sharded clusters specifically:

- *Backup and Restore Sharded Clusters* (page 265)
- *Restore a Sharded Cluster* (page 272)
- *MongoDB Backup Methods* (page 192)

Procedure Always restore *sharded clusters* as a whole. When you restore a single shard, keep in mind that the *balancer* process might have moved *chunks* to or from this shard since the last backup. If that's the case, you must manually move those chunks, as described in this procedure.

Step 1: Restore the shard as you would any other `mongod` instance. See *MongoDB Backup Methods* (page 192) for overviews of these procedures.

Step 2: Manage the chunks. For all chunks that migrate away from this shard, you do not need to do anything at this time. You do not need to delete these documents from the shard because the chunks are automatically filtered out from queries by `mongos`. You can remove these documents from the shard, if you like, at your leisure.

For chunks that migrate to this shard after the most recent backup, you must manually recover the chunks using backups of other shards, or some other source. To determine what chunks have moved, view the `changelog` collection in the *Config Database* (page 754).

Restore a Sharded Cluster

On this page

Overview You can restore a sharded cluster either from *snapshots* (page 256) or from *BSON database dumps* (page 269) created by the `mongodump` tool. This document provides procedures for both:

- *Restore a Sharded Cluster with Filesystem Snapshots* (page 273)
- *Restore a Sharded Cluster with Database Dumps* (page 273)

Related Documents For an overview of backups in MongoDB, see *MongoDB Backup Methods* (page 192). For complete information on backups and backups of sharded clusters in particular, see *Backup and Restore Sharded Clusters* (page 265).

For backup procedures, see:

- *Backup a Sharded Cluster with Filesystem Snapshots* (page 267)
- *Backup a Sharded Cluster with Database Dumps* (page 269)

Procedures Use the procedure for the type of backup files to restore.

Restore a Sharded Cluster with Filesystem Snapshots

Step 1: Shut down the entire cluster. Stop all `mongos` and `mongod` processes, including all shards *and* all config servers.

Connect to each member use the following operation:

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

For version 2.4 or earlier, use `db.shutdownServer({force:true})`.

Step 2: Restore the data files. One each server, extract the data files to the location where the `mongod` instance will access them. 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 258) and *Restore a Database with mongorestore* (page 264) sections for details on these procedures.

Data files for each config server.

Step 3: Restart the config servers. Restart each *config server* (page 684) `mongod` instance by issuing a command similar to the following for each, using values appropriate to your configuration:

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

Step 4: If shard hostnames have changed, update the config string and config database. If shard hostnames **have changed**, start **one** `mongos` instance using the updated config string with the new `configdb` hostnames and ports.

Then update the `shards` collection in the *Config Database* (page 754) to reflect the new hostnames. Then stop the `mongos` instance.

Step 5: Restart all the shard `mongod` instances.

Step 6: Restart all the `mongos` instances. If shard hostnames **have changed**, make sure to use the updated config string.

Step 7: Connect to a `mongos` to ensure the cluster is operational. Connect to a `mongos` instance from a `mongo` shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

Restore a Sharded Cluster with Database Dumps

Step 1: Shut down the entire cluster. Stop all `mongos` and `mongod` processes, including all shards *and* all config servers.

Connect to each member use the following operation:

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

For version 2.4 or earlier, use `db.shutdownServer({force:true})`.

Step 2: Restore the data files. One each server, use `mongorestore` to restore the database dump to the location where the `mongod` instance will access the data.

The following example restores a database dump located at `/opt/backup/` to the `/data/` directory. This requires that there are no active `mongod` instances attached to the `/data` directory.

```
mongorestore --dbpath /data /opt/backup
```

Step 3: Restart the config servers. Restart each *config server* (page 684) `mongod` instance by issuing a command similar to the following for each, using values appropriate to your configuration:

```
mongod --configsvr --dbpath /data/configdb --port 27019
```

Step 4: If shard hostnames have changed, update the config string and config database. If shard hostnames **have changed**, start **one** `mongos` instance using the updated config string with the new `configdb` hostnames and ports.

Then update the `shards` collection in the *Config Database* (page 754) to reflect the new hostnames. Then stop the `mongos` instance.

Step 5: Restart all the shard `mongod` instances.

Step 6: Restart all the `mongos` instances. If shard hostnames **have changed**, make sure to use the updated config string.

Step 7: Connect to a `mongos` to ensure the cluster is operational. Connect to a `mongos` instance from a `mongo` shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

Recover Data after an Unexpected Shutdown

On this page

If MongoDB does not shutdown cleanly [90] the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption. [91]

To prevent data inconsistency and corruption, always shut down the database cleanly and use the *durability journaling*. MongoDB writes data to the journal, by default, every 100 milliseconds, such that MongoDB can always recover to a consistent state even in the case of an unclean shutdown due to power loss or other system failure.

If you are *not* running as part of a *replica set* **and** do *not* have journaling enabled, use the following procedure to recover data that may be in an inconsistent state. If you are running as part of a replica set, you should *always* restore from a backup or restart the `mongod` instance with an empty `dbPath` and allow MongoDB to perform an initial sync to restore the data.

See also:

The *Administration* (page 191) documents, including *Replica Set Syncing* (page 596), and the documentation on the `--repair` `repairPath` and `storage.journal.enabled` settings.

Process

Indications When you are aware of a `mongod` instance running without journaling that stops unexpectedly **and** you're not running with replication, you should always run the repair operation before starting MongoDB again. If you're using replication, then restore from a backup and allow replication to perform an initial *sync* (page 596) to restore data.

If the `mongod.lock` file in the data directory specified by `dbPath`, `/data/db` by default, is *not* a zero-byte file, then `mongod` will refuse to start, and you will find a message that contains the following line in your MongoDB log our output:

```
Unclean shutdown detected.
```

This indicates that you need to run `mongod` with the `--repair` option. If you run repair when the `mongodb.lock` file exists in your `dbPath`, or the optional `--repairpath`, you will see a message that contains the following line:

```
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 192) 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 640).

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

- Use the `--repair` option in conjunction with the `--repairpath` option. `mongod` will read the existing data files, and write the existing data to new data files.

  You do not need to remove the `mongod.lock` file before using this procedure.

[90] To ensure a clean shut down, use the `db.shutdownServer()` from the `mongo` shell, your control script, the `mongod --shutdown` option on Linux systems, "Control-C" when running `mongod` in interactive mode, or `kill $(pidof mongod)` or `kill -2 $(pidof mongod)`.

[91] You can also use the `db.collection.validate()` method to test the integrity of a single collection. However, this process is time consuming, and without journaling you can safely assume that the data is in an invalid state and you should either run the repair operation or resync from an intact member of the replica set.

- Use the *--repair* option. mongod will read the existing data files, write the existing data to new files and replace the existing, possibly corrupt, files with new files.

  You must remove the mongod.lock file before using this procedure.

Note: *--repair* functionality is also available in the shell with the db.repairDatabase() helper for the repairDatabase command.

Procedures

Important: Always Run mongod as the same user to avoid changing the permissions of the MongoDB data files.

Repair Data Files and Preserve Original Files To repair your data files using the *--repairpath* option to preserve the original data files unmodified.

Repair Data Files without Preserving Original Files To repair your data files without preserving the original files, do not use the *--repairpath* option, as in the following procedure:

Warning: After you remove the mongod.lock file you *must* run the *--repair* process before using your database.

Step 1: Start mongod using the option to replace the original files with the repaired files. Start the mongod instance using the *--repair* option **and** the *--repairpath* option. Issue a command similar to the following:

```
mongod --dbpath /data/db --repair --repairpath /data/db0
```

When this completes, the new repaired data files will be in the /data/db0 directory.

Step 2: Start mongod with the new data directory. Start mongod using the following invocation to point the dbPath at /data/db0:

```
mongod --dbpath /data/db0
```

Once you confirm that the data files are operational you may delete or archive the old data files in the /data/db directory. You may also wish to move the repaired files to the old database location or update the dbPath to indicate the new location.

Step 1: Remove the stale lock file. For example:

```
rm /data/db/mongod.lock
```

Replace /data/db with your dbPath where your MongoDB instance's data files reside.

Step 2: Start mongod using the option to replace the original files with the repaired files. Start the mongod instance using the *--repair* option, which replaces the original data files with the repaired data files. Issue a command similar to the following:

```
mongod --dbpath /data/db --repair
```

When this completes, the repaired data files will replace the original data files in the /data/db directory.

Step 3: Start `mongod` as usual. Start `mongod` using the following invocation to point the `dbPath` at `/data/db`:

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

`mongod.lock`

In normal operation, you should **never** remove the `mongod.lock` file and start `mongod`. Instead consider the one of the above methods to recover the database and remove the lock files. In dire situations you can remove the lockfile, and start the database using the possibly corrupt files, and attempt to recover data from the database; however, it's impossible to predict the state of the database in these situations.

If you are not running with journaling, and your database shuts down unexpectedly for *any* reason, you should always proceed *as if* your database is in an inconsistent and likely corrupt state. If at all possible restore from *backup* (page 192) 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 640).

5.2.3 MongoDB Scripting

The `mongo` shell is an interactive JavaScript shell for MongoDB, and is part of all MongoDB distributions[92]. This section provides an introduction to the shell, and outlines key functions, operations, and use of the `mongo` shell. Also consider *FAQ: The mongo Shell* (page 775) and the `shell method` and other relevant `reference material`.

Note: Most examples in the `MongoDB Manual` use the `mongo` shell; however, many `drivers` provide similar interfaces to MongoDB.

Server-side JavaScript **(page 277)** Details MongoDB's support for executing JavaScript code for server-side operations.

Data Types in the mongo Shell **(page 279)** Describes the super-set of JSON available for use in the `mongo` shell.

Write Scripts for the mongo Shell **(page 282)** An introduction to the `mongo` shell for writing scripts to manipulate data and administer MongoDB.

Getting Started with the mongo Shell **(page 284)** Introduces the use and operation of the MongoDB shell.

Access the mongo Shell Help Information **(page 288)** Describes the available methods for accessing online help for the operation of the `mongo` interactive shell.

mongo Shell Quick Reference **(page 290)** A high level reference to the use and operation of the `mongo` shell.

Server-side JavaScript

On this page

[92] http://www.mongodb.org/downloads

Overview

MongoDB provides the following commands, methods, and operator that perform server-side execution of JavaScript code:

- `mapReduce` and the corresponding `mongo` shell method `db.collection.mapReduce()`. `mapReduce` operations *map*, or associate, values to keys, and for keys with multiple values, *reduce* the values for each key to a single object. For more information, see *Map-Reduce* (page 442).
- `eval` command and the corresponding `mongo` shell method `db.eval()`. `eval` operations evaluates JavaScript functions on the database server. You cannot use the `eval` command and `db.eval()` method with sharded collections. For replica sets, you can only run the `eval` command and `db.eval()` method against the primary. For more information, see `eval` command and `db.eval()` method reference pages.
- `$where` operator that evaluates a JavaScript expression or a function in order to query for documents.

You can also specify a JavaScript file to the `mongo` shell to run on the server. For more information, see *Running .js files via a mongo shell Instance on the Server* (page 278)

JavaScript in MongoDB

Although the aforementioned operations use JavaScript, most interactions with MongoDB do not use JavaScript but use an `idiomatic driver` in the language of the interacting application.

You can also disable server-side execution of JavaScript. For details, see *Disable Server-Side Execution of JavaScript* (page 279).

Running `.js` files via a `mongo` shell Instance on the Server

You can specify a JavaScript (`.js`) file to a `mongo` shell instance to execute the file on the server. This is a good technique for performing batch administrative work. When you run `mongo` shell on the server, connecting via the localhost interface, the connection is fast with low latency.

The *command helpers* (page 291) provided in the `mongo` shell are not available in JavaScript files because they are not valid JavaScript. The following table maps the most common `mongo` shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.getUsers()`
`show roles`	`db.getRoles({showBuiltinRoles: true})`
`show log <logname>`	`db.adminCommand({ 'getLog' : '<logname>' })`
`show logs`	`db.adminCommand({ 'getLog' : '*' })`

`it`	`cursor = db.collection.find()`
	`if ( cursor.hasNext() ){`
	`  cursor.next();`
	`}`

Concurrency

Changed in version 2.4.

The V8 JavaScript engine, which became the default in 2.4, allows multiple JavaScript operations to execute at the same time. Prior to 2.4, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` could only run a single JavaScript operation at a time.

Refer to the individual method or operator documentation for any concurrency information. See also the *concurrency table* (page 779).

Disable Server-Side Execution of JavaScript

You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting `security.javascriptEnabled` in a configuration file.

See also:

Store a JavaScript Function on the Server (page 247)

Data Types in the `mongo` Shell

On this page

MongoDB *BSON* provides support for additional data types than *JSON*. Drivers provide native support for these data types in host languages and the mongo shell also provides several helper classes to support the use of these data types in the mongo JavaScript shell. See the Extended JSON reference for additional information.

Types

Date The mongo shell provides various methods to return the date, either as a string or as a Date object:

- Date() method which returns the current date as a string.
- new Date() constructor which returns a Date object using the ISODate() wrapper.
- ISODate() constructor which returns a Date object using the ISODate() wrapper.

Internally, *Date* (page 189) objects are stored as a 64 bit integer representing the number of milliseconds since the Unix epoch (Jan 1, 1970), which results in a representable date range of about 290 millions years into the past and future.

Return Date as a String To return the date as a string, use the Date() method, as in the following example:

```
var myDateString = Date();
```

To print the value of the variable, type the variable name in the shell, as in the following:

```
myDateString
```

The result is the value of myDateString:

```
Wed Dec 19 2012 01:03:25 GMT-0500 (EST)
```

To verify the type, use the typeof operator, as in the following:

```
typeof myDateString
```

The operation returns string.

Return Date The mongo shell wraps objects of Date type with the ISODate helper; however, the objects remain of type Date.

The following example uses both the new Date() constructor and the ISODate() constructor to return Date objects.

```
var myDate = new Date();
var myDateInitUsingISODateWrapper = ISODate();
```

You can use the new operator with the ISODate() constructor as well.

To print the value of the variable, type the variable name in the shell, as in the following:

```
myDate
```

The result is the Date value of myDate wrapped in the ISODate() helper:

```
ISODate("2012-12-19T06:01:17.171Z")
```

To verify the type, use the instanceof operator, as in the following:

```
myDate instanceof Date
myDateInitUsingISODateWrapper instanceof Date
```

The operation returns `true` for both.

ObjectId The `mongo` shell provides the `ObjectId()` wrapper class around the *ObjectId* data type. To generate a new ObjectId, use the following operation in the `mongo` shell:

```
new ObjectId
```

See

ObjectId (page 184) for full documentation of ObjectIds in MongoDB.

NumberLong By default, the `mongo` shell treats all numbers as floating-point values. The `mongo` shell provides the `NumberLong()` wrapper to handle 64-bit integers.

The `NumberLong()` wrapper accepts the long as a string:

```
NumberLong("2090845886852")
```

The following examples use the `NumberLong()` wrapper to write to the collection:

```
db.collection.insert( { _id: 10, calc: NumberLong("2090845886852") } )
db.collection.update( { _id: 10 },
                      { $set:  { calc: NumberLong("2555555000000") } } )
db.collection.update( { _id: 10 },
                      { $inc: { calc: NumberLong(5) } } )
```

Retrieve the document to verify:

```
db.collection.findOne( { _id: 10 } )
```

In the returned document, the `calc` field contains a `NumberLong` object:

```
{ "_id" : 10, "calc" : NumberLong("2555555000005") }
```

If you use the `$inc` to increment the value of a field that contains a `NumberLong` object by a **float**, the data type changes to a floating point value, as in the following example:

1. Use `$inc` to increment the `calc` field by `5`, which the `mongo` shell treats as a float:

   ```
   db.collection.update( { _id: 10 },
                         { $inc: { calc: 5 } } )
   ```

2. Retrieve the updated document:

   ```
   db.collection.findOne( { _id: 10 } )
   ```

   In the updated document, the `calc` field contains a floating point value:

   ```
   { "_id" : 10, "calc" : 2555555000010 }
   ```

NumberInt By default, the `mongo` shell treats all numbers as floating-point values. The `mongo` shell provides the `NumberInt()` constructor to explicitly specify 32-bit integers.

Check Types in the `mongo` Shell

To determine the type of fields, the `mongo` shell provides the `instanceof` and `typeof` operators.

instanceof `instanceof` returns a boolean to test if a value is an instance of some type.

For example, the following operation tests whether the `_id` field is an instance of type `ObjectId`:

```
mydoc._id instanceof ObjectId
```

The operation returns `true`.

typeof `typeof` returns the type of a field.

For example, the following operation returns the type of the `_id` field:

```
typeof mydoc._id
```

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

Write Scripts for the mongo Shell

On this page

You can write scripts for the `mongo` shell in JavaScript that manipulate data in MongoDB or perform administrative operation. For more information about the `mongo` shell see *MongoDB Scripting* (page 277), and see the *Running .js files via a mongo shell Instance on the Server* (page 278) section for more information about using these `mongo` script.

This tutorial provides an introduction to writing JavaScript that uses the `mongo` shell to access MongoDB.

Opening New Connections

From the `mongo` shell or from a JavaScript file, you can instantiate database connections using the `Mongo()` constructor:

```
new Mongo()
new Mongo(<host>)
new Mongo(<host:port>)
```

Consider the following example that instantiates a new connection to the MongoDB instance running on localhost on the default port and sets the global `db` variable to `myDatabase` using the `getDB()` method:

```
conn = new Mongo();
db = conn.getDB("myDatabase");
```

Additionally, you can use the `connect()` method to connect to the MongoDB instance. The following example connects to the MongoDB instance that is running on `localhost` with the non-default port `27020` and set the global `db` variable:

```
db = connect("localhost:27020/myDatabase");
```

Differences Between Interactive and Scripted mongo

When writing scripts for the `mongo` shell, consider the following:

- To set the `db` global variable, use the `getDB()` method or the `connect()` method. You can assign the database reference to a variable other than `db`.
- Write operations in the `mongo` shell use the "safe writes" by default. If performing bulk operations, use the `Bulk()` methods. See *Write Method Acknowledgements* (page 838) for more information.

  Changed in version 2.6: Before MongoDB 2.6, call `db.getLastError()` explicitly to wait for the result of *write operations* (page 77).
- You **cannot** use any shell helper (e.g. `use <dbname>`, `show dbs`, etc.) inside the JavaScript file because they are not valid JavaScript.

  The following table maps the most common `mongo` shell helpers to their JavaScript equivalents.

Shell Helpers	JavaScript Equivalents
`show dbs`, `show databases`	`db.adminCommand('listDatabases')`
`use <db>`	`db = db.getSiblingDB('<db>')`
`show collections`	`db.getCollectionNames()`
`show users`	`db.getUsers()`
`show roles`	`db.getRoles({showBuiltinRoles: true})`
`show log <logname>`	`db.adminCommand({ 'getLog' : '<logname>' })`
`show logs`	`db.adminCommand({ 'getLog' : '*' })`

`it`	`cursor = db.collection.find()`
	`if ( cursor.hasNext() ){`
	`  cursor.next();`
	`}`

- In interactive mode, `mongo` prints the results of operations including the content of all cursors. In scripts, either use the JavaScript `print()` function or the `mongo` specific `printjson()` function which returns formatted JSON.

Example

To print all items in a result cursor in `mongo` shell scripts, use the following idiom:

```
cursor = db.collection.find();
while ( cursor.hasNext() ) {
   printjson( cursor.next() );
}
```

Scripting

From the system prompt, use `mongo` to evaluate JavaScript.

`--eval` option Use the *--eval* option to `mongo` to pass the shell a JavaScript fragment, as in the following:

```
mongo test --eval "printjson(db.getCollectionNames())"
```

This returns the output of `db.getCollectionNames()` using the `mongo` shell connected to the `mongod` or `mongos` instance running on port `27017` on the `localhost` interface.

Execute a JavaScript file You can specify a `.js` file to the `mongo` shell, and `mongo` will execute the JavaScript directly. Consider the following example:

```
mongo localhost:27017/test myjsfile.js
```

This operation executes the `myjsfile.js` script in a `mongo` shell that connects to the `test` *database* on the `mongod` instance accessible via the `localhost` interface on port `27017`.

Alternately, you can specify the mongodb connection parameters inside of the javascript file using the `Mongo()` constructor. See *Opening New Connections* (page 282) for more information.

You can execute a `.js` file from within the `mongo` shell, using the `load()` function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the `myjstest.js` file.

The `load()` method accepts relative and absolute paths. If the current working directory of the `mongo` shell is `/data/db`, and the `myjstest.js` resides in the `/data/db/scripts` directory, then the following calls within the `mongo` shell would be equivalent:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

Note: There is no search path for the `load()` function. If the desired script is not in the current working directory or the full specified path, `mongo` will not be able to access the file.

Getting Started with the `mongo` Shell

On this page

This document provides a basic introduction to using the `mongo` shell. See *Install MongoDB* (page 5) for instructions on installing MongoDB for your system.

Start the `mongo` Shell

To start the `mongo` shell and connect to your `MongoDB` instance running on **localhost** with **default port**:

1. Go to your `<mongodb installation dir>`:

```
cd <mongodb installation dir>
```

2. Type `./bin/mongo` to start `mongo`:

   ```
   ./bin/mongo
   ```

   If you have added the `<mongodb installation dir>/bin` to the `PATH` environment variable, you can just type `mongo` instead of `./bin/mongo`.

3. To display the database you are using, type `db`:

   ```
   db
   ```

   The operation should return `test`, which is the default database. To switch databases, issue the `use <db>` helper, as in the following example:

   ```
   use <database>
   ```

   To list the available databases, use the helper `show dbs`. See also *How can I access different databases temporarily?* (page 775) to access a different database from the current database without switching your current database context (i.e. `db..`)

To start the `mongo` shell with other options, see *examples of starting up mongo* and `mongo reference` which provides details on the available options.

Note: When starting, `mongo` checks the user's `HOME` directory for a JavaScript file named *.mongorc.js*. If found, `mongo` interprets the content of `.mongorc.js` before displaying the prompt for the first time. If you use the shell to evaluate a JavaScript file or expression, either by using the `--eval` option on the command line or by specifying *a .js file to mongo*, `mongo` will read the `.mongorc.js` file *after* the JavaScript has finished processing. You can prevent `.mongorc.js` from being loaded by using the `--norc` option.

Executing Queries

From the `mongo` shell, you can use the `shell methods` to run queries, as in the following example:

```
db.<collection>.find()
```

- The `db` refers to the current database.
- The `<collection>` is the name of the collection to query. See *Collection Help* (page 289) to list the available collections.

  If the `mongo` shell does not accept the name of the collection, for instance if the name contains a space, hyphen, or starts with a number, you can use an alternate syntax to refer to the collection, as in the following:

  ```
  db["3test"].find()
  ```

  ```
  db.getCollection("3test").find()
  ```

- The `find()` method is the JavaScript method to retrieve documents from `<collection>`. The `find()` method returns a *cursor* to the results; however, in the `mongo` shell, if the returned cursor is not assigned to a variable using the `var` keyword, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents that match the query. The `mongo` shell will prompt `Type it` to iterate another 20 times.

  You can set the `DBQuery.shellBatchSize` attribute to change the number of iteration from the default value `20`, as in the following example which sets it to `10`:

```
DBQuery.shellBatchSize = 10;
```

For more information and examples on cursor handling in the mongo shell, see *Cursors* (page 68).

See also *Cursor Help* (page 289) for list of cursor help in the mongo shell.

For more documentation of basic MongoDB operations in the mongo shell, see:

- *Getting Started with MongoDB* (page 52)
- *mongo Shell Quick Reference* (page 290)
- *Read Operations* (page 64)
- *Write Operations* (page 77)
- *Indexing Tutorials* (page 519)

Print

The mongo shell automatically prints the results of the find() method if the returned cursor is not assigned to a variable using the var keyword. To format the result, you can add the .pretty() to the operation, as in the following:

```
db.<collection>.find().pretty()
```

In addition, you can use the following explicit print methods in the mongo shell:

- print() to print without formatting
- print(tojson(<obj>)) to print with *JSON* formatting and equivalent to printjson()
- printjson() to print with *JSON* formatting and equivalent to print(tojson(<obj>))

Evaluate a JavaScript File

You can execute a .js file from within the mongo shell, using the load() function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the myjstest.js file.

The load() method accepts relative and absolute paths. If the current working directory of the mongo shell is /data/db, and the myjstest.js resides in the /data/db/scripts directory, then the following calls within the mongo shell would be equivalent:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

Note: There is no search path for the load() function. If the desired script is not in the current working directory or the full specified path, mongo will not be able to access the file.

Use a Custom Prompt

You may modify the content of the prompt by creating the variable prompt in the shell. The prompt variable can hold strings as well as any arbitrary JavaScript. If prompt holds a function that returns a string, mongo can display dynamic information in each prompt. Consider the following examples:

Example

Create a prompt with the number of operations issued in the current session, define the following variables:

```
cmdCount = 1;
prompt = function() {
             return (cmdCount++) + "> ";
         }
```

The prompt would then resemble the following:

```
1> db.collection.find()
2> show collections
3>
```

Example

To create a `mongo` shell prompt in the form of `<database>@<hostname>$` define the following variables:

```
host = db.serverStatus().host;

prompt = function() {
             return db+"@"+host+"$ ";
         }
```

The prompt would then resemble the following:

```
<database>@<hostname>$ use records
switched to db records
records@<hostname>$
```

Example

To create a `mongo` shell prompt that contains the system up time *and* the number of documents in the current database, define the following prompt variable:

```
prompt = function() {
             return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > ";
         }
```

The prompt would then resemble the following:

```
Uptime:5897 Documents:6 > db.people.save({name : "James"});
Uptime:5948 Documents:7 >
```

Use an External Editor in the `mongo` Shell

New in version 2.2.

In the `mongo` shell you can use the `edit` operation to edit a function or variable in an external editor. The `edit` operation uses the value of your environments `EDITOR` variable.

At your system prompt you can define the `EDITOR` variable and start `mongo` with the following two operations:

```
export EDITOR=vim
mongo
```

Then, consider the following example shell session:

```
MongoDB shell version: 2.2.0
> function f() {}
> edit f
> f
function f() {
    print("this really works");
}
> f()
this really works
> o = {}
{ }
> edit o
> o
{ "soDoes" : "this" }
>
```

Note: As `mongo` shell interprets code edited in an external editor, it may modify code in functions, depending on the JavaScript compiler. For `mongo` may convert `1+1` to `2` or remove comments. The actual changes affect only the appearance of the code and will vary based on the version of JavaScript used but will not affect the semantics of the code.

Exit the Shell

To exit the shell, type `quit()` or use the `<Ctrl-c>` shortcut.

Access the `mongo` Shell Help Information

On this page

In addition to the documentation in the `MongoDB Manual`, the `mongo` shell provides some additional information in its "online" help system. This document provides an overview of accessing this help information.

See also:

- `mongo Manual Page`
- *MongoDB Scripting* (page 277), and
- *mongo Shell Quick Reference* (page 290).

Command Line Help

To see the list of options and help for starting the `mongo` shell, use the *`--help`* option from the command line:

```
mongo --help
```

Shell Help

To see the list of help, in the `mongo` shell, type `help`:

```
help
```

Database Help

- To see the list of databases on the server, use the `show dbs` command:

  ```
  show dbs
  ```

  New in version 2.4: `show databases` is now an alias for `show dbs`

- To see the list of help for methods you can use on the `db` object, call the `db.help()` method:

  ```
  db.help()
  ```

- To see the implementation of a method in the shell, type the `db.<method name>` without the parenthesis (`()`), as in the following example which will return the implementation of the method `db.addUser()`:

  ```
  db.addUser
  ```

Collection Help

- To see the list of collections in the current database, use the `show collections` command:

  ```
  show collections
  ```

- To see the help for methods available on the collection objects (e.g. `db.<collection>`), use the `db.<collection>.help()` method:

  ```
  db.collection.help()
  ```

  `<collection>` can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the collection method implementation, type the `db.<collection>.<method>` name without the parenthesis (`()`), as in the following example which will return the implementation of the `save()` method:

  ```
  db.collection.save
  ```

Cursor Help

When you perform *read operations* (page 65) with the `find()` method in the `mongo` shell, you can use various cursor methods to modify the `find()` behavior and various JavaScript methods to handle the cursor returned from the `find()` method.

- To list the available modifier and cursor handling methods, use the `db.collection.find().help()` command:

```
db.collection.find().help()
```

<collection> can be the name of a collection that exists, although you may specify a collection that doesn't exist.

- To see the implementation of the cursor method, type the db.<collection>.find().<method> name without the parenthesis (()), as in the following example which will return the implementation of the toArray() method:

```
db.collection.find().toArray
```

Some useful methods for handling cursors are:

- hasNext() which checks whether the cursor has more documents to return.
- next() which returns the next document and advances the cursor position forward by one.
- forEach(<function>) which iterates the whole cursor and applies the <function> to each document returned by the cursor. The <function> expects a single argument which corresponds to the document from each iteration.

For examples on iterating a cursor and retrieving the documents from the cursor, see *cursor handling* (page 68). See also *js-query-cursor-methods* for all available cursor methods.

Type Help

To get a list of the wrapper classes available in the mongo shell, such as BinData(), type help misc in the mongo shell:

```
help misc
```

mongo Shell Quick Reference

On this page

mongo Shell Command History

You can retrieve previous commands issued in the mongo shell with the up and down arrow keys. Command history is stored in ~/.dbshell file. See *.dbshell* for more information.

Command Line Options

The `mongo` executable can be started with numerous options. See `mongo executable` page for details on all available options.

The following table displays some common options for `mongo`:

Option	Description
`--help`	Show command line options
`--nodb`	Start `mongo` shell without connecting to a database. To connect later, see *Opening New Connections* (page 282).
`--shell`	Used in conjunction with a JavaScript file (i.e. <*file.js*>) to continue in the `mongo` shell after running the JavaScript file. See *JavaScript file* (page 284) for an example.

Command Helpers

The `mongo` shell provides various help. The following table displays some common help methods and commands:

Help Methods and Commands	Description
`help`	Show help.
`db.help()`	Show help for database methods.
`db.<collection>.help()`	Show help on collection methods. The `<collection>` can be the name of an existing collection or a non-existing collection.
`show dbs`	Print a list of all databases on the server.
`use <db>`	Switch current database to `<db>`. The `mongo` shell variable `db` is set to the current database.
`show collections`	Print a list of all collections for current database
`show users`	Print a list of users for current database.
`show roles`	Print a list of all roles, both user-defined and built-in, for the current database.
`show profile`	Print the five most recent operations that took 1 millisecond or more. See documentation on the *database profiler* (page 239) for more information.
`show databases`	New in version 2.4: Print a list of all available databases.
`load()`	Execute a JavaScript file. See *Getting Started with the mongo Shell* (page 284) for more information.

Basic Shell JavaScript Operations

The `mongo` shell provides a `JavaScript API` for database operations.

In the `mongo` shell, `db` is the variable that references the current database. The variable is automatically set to the default database `test` or is set when you use the `use <db>` to switch current database.

The following table displays some common JavaScript operations:

JavaScript Database Operations	Description
`db.auth()`	If running in secure mode, authenticate the user.

`coll = db.<collection>`	Set a specific collection in the current database to a variable `coll`, as in the following example:
	`coll = db.myCollection;`
	You can perform operations on the `myCollection` using the variable, as in the following example:
	`coll.find();`

`find()`	Find all documents in the collection and returns a cursor. See the `db.collection.find()` and *Query Documents* (page 100) for more information and examples. See *Cursors* (page 68) for additional information on cursor handling in the `mongo` shell.
`insert()`	Insert a new document into the collection.
`update()`	Update an existing document in the collection. See *Write Operations* (page 77) for more information.
`save()`	Insert either a new document or update an existing document in the collection. See *Write Operations* (page 77) for more information.
`remove()`	Delete documents from the collection. See *Write Operations* (page 77) for more information.
`drop()`	Drops or removes completely the collection.
`ensureIndex()`	Create a new index on the collection if the index does not exist; otherwise, the operation has no effect.
`db.getSiblingDB()`	Return a reference to another database using this same connection without explicitly switching the current database. This allows for cross database queries. See *How can I access different databases temporarily?* (page 775) for more information.

For more information on performing operations in the shell, see:

- *MongoDB CRUD Concepts* (page 64)
- *Read Operations* (page 64)
- *Write Operations* (page 77)
- *js-administrative-methods*

Keyboard Shortcuts

Changed in version 2.2.

The `mongo` shell provides most keyboard shortcuts similar to those found in the `bash` shell or in Emacs. For some functions `mongo` provides multiple key bindings, to accommodate several familiar paradigms.

The following table enumerates the keystrokes supported by the `mongo` shell:

Keystroke	Function
Up-arrow	previous-history
Down-arrow	next-history
Home	beginning-of-line
End	end-of-line
Tab	autocomplete

Continued on next page

Table 5.1 – continued from previous page

Keystroke	Function
Left-arrow	backward-character
Right-arrow	forward-character
Ctrl-left-arrow	backward-word
Ctrl-right-arrow	forward-word
Meta-left-arrow	backward-word
Meta-right-arrow	forward-word
Ctrl-A	beginning-of-line
Ctrl-B	backward-char
Ctrl-C	exit-shell
Ctrl-D	delete-char (or exit shell)
Ctrl-E	end-of-line
Ctrl-F	forward-char
Ctrl-G	abort
Ctrl-J	accept-line
Ctrl-K	kill-line
Ctrl-L	clear-screen
Ctrl-M	accept-line
Ctrl-N	next-history
Ctrl-P	previous-history
Ctrl-R	reverse-search-history
Ctrl-S	forward-search-history
Ctrl-T	transpose-chars
Ctrl-U	unix-line-discard
Ctrl-W	unix-word-rubout
Ctrl-Y	yank
Ctrl-Z	Suspend (job control works in linux)
Ctrl-H (i.e. Backspace)	backward-delete-char
Ctrl-I (i.e. Tab)	complete
Meta-B	backward-word
Meta-C	capitalize-word
Meta-D	kill-word
Meta-F	forward-word
Meta-L	downcase-word
Meta-U	upcase-word
Meta-Y	yank-pop
Meta-[Backspace]	backward-kill-word
Meta-<	beginning-of-history
Meta->	end-of-history

Queries

In the `mongo` shell, perform read operations using the `find()` and `findOne()` methods.

The `find()` method returns a cursor object which the `mongo` shell iterates to print documents on screen. By default, `mongo` prints the first 20. The `mongo` shell will prompt the user to "`Type it`" to continue iterating the next 20 results.

The following table provides some common read operations in the `mongo` shell:

Read Operations	Description

`db.collection.find(<query>)`	Find the documents matching the `<query>` criteria in the collection. If the `<query>` criteria is not specified or is empty (i.e `{}`), the read operation selects all documents in the collection.
	The following example selects the documents in the `users` collection with the `name` field equal to `"Joe"`:
	`coll = db.users;`
	`coll.find( { name: "Joe" } );`
	For more information on specifying the `<query>` criteria, see *Query Documents* (page 100).

`db.collection.find( <query>, <projection> )`	Find documents matching the `<query>` criteria and return just specific fields in the `<projection>`.
	The following example selects all documents from the collection but returns only the `name` field and the `_id` field. The `_id` is always returned unless explicitly specified to not return.
	`coll = db.users;`
	`coll.find( { },`
	`{ name: true }`
	`);`
	For more information on specifying the `<projection>`, see *Limit Fields to Return from a Query* (page 112).

`db.collection.find().sort( <sort order> )`	Return results in the specified `<sort order>`.
	The following example selects all documents from the collection and returns the results sorted by the `name` field in ascending order (`1`). Use `-1` for descending order:
	`coll = db.users;`
	`coll.find().sort( { name: 1 } );`

`db.collection.find( <query> ).sort( <sort order> )`	Return the documents matching the `<query>` criteria in the specified `<sort order>`.
`db.collection.find( ... ).limit( <n> )`	Limit result to `<n>` rows. Highly recommended if you need only a certain number of rows for best performance.
`db.collection.find( ... ).skip( <n> )`	Skip `<n>` results.
`count()`	Returns total number of documents in the collection.
`db.collection.find( <query> ).count()`	Returns the total number of documents that match the query. The `count()` ignores `limit()` and `skip()`. For example, if 100 records match but the limit is 10, `count()` will return 100. This will be faster than iterating yourself, but still take time.

`db.collection.findOne( <query> )`	Find and return a single document. Returns null if not found.
	The following example selects a single document in the `users` collection with the `name` field matches to `"Joe"`:
	`coll = db.users;`
	`coll.findOne( { name: "Joe" } );`
	Internally, the `findOne()` method is the `find()` method with a `limit(1)`.

See *Query Documents* (page 100) and *Read Operations* (page 64) documentation for more information and examples. See `http://docs.mongodb.org/manual/reference/operator/query` to specify other query operators.

Error Checking Methods

Changed in version 2.6.

The `mongo` shell write methods now integrates the *Write Concern* (page 82) directly into the method execution rather than with a separate `db.getLastError()` method. As such, the write methods now return a `WriteResult()` object that contains the results of the operation, including any write errors and write concern errors.

Previous versions used `db.getLastError()` and `db.getLastErrorObj()` methods to return error information.

Administrative Command Helpers

The following table lists some common methods to support database administration:

JavaScript Database Administration Methods	Description
`db.cloneDatabase(<host>)`	Clone the current database from the `<host>` specified. The `<host>` database instance must be in noauth mode.
`db.copyDatabase(<from>, <to>, <host>)`	Copy the `<from>` database from the `<host>` to the `<to>` database on the current server. The `<host>` database instance must be in `noauth` mode.
`db.fromColl.renameCollection(<toColl>)`	Rename collection from `fromColl` to `<toColl>`.
`db.repairDatabase()`	Repair and compact the current database. This operation can be very slow on large databases.
`db.addUser( <user>, <pwd> )`	Add user to current database.
`db.getCollectionNames()`	Get the list of all collections in the current database.
`db.dropDatabase()`	Drops the current database.

See also *administrative database methods* for a full list of methods.

Opening Additional Connections

You can create new connections within the `mongo` shell.

The following table displays the methods to create the connections:

JavaScript Connection Create Methods	Description
`db = connect("<host><:port>/<dbname>")`	Open a new database connection.
`conn = new Mongo()` `db = conn.getDB("dbname")`	Open a connection to a new server using `new Mongo()`. Use `getDB()` method of the connection to select a database.

See also *Opening New Connections* (page 282) for more information on the opening new connections from the `mongo` shell.

Miscellaneous

The following table displays some miscellaneous methods:

Method	Description
`Object.bsonsize(<document>)`	Prints the *BSON* size of a <document> in bytes

See the MongoDB JavaScript API Documentation[93] for a full list of JavaScript methods .

Additional Resources

Consider the following reference material that addresses the `mongo` shell and its interface:

- `mongo`
- *js-administrative-methods*
- *database-commands*
- *Aggregation Reference* (page 470)

Additionally, the MongoDB source code repository includes a jstests directory[94] which contains numerous `mongo` shell scripts.

5.2.4 MongoDB Tutorials

This page lists the tutorials available as part of the `MongoDB Manual`. In addition to these documents, you can refer to the introductory *MongoDB Tutorial* (page 52). If there is a process or pattern that you would like to see included here, please open a Jira Case[95].

Getting Started

- *Install MongoDB on Linux Systems* (page 16)
- *Install MongoDB on Red Hat Enterprise or CentOS Linux* (page 6)
- *Install MongoDB on Debian* (page 13)
- *Install MongoDB on Ubuntu* (page 10)
- *Install MongoDB on OS X* (page 19)
- *Install MongoDB on Windows* (page 21)
- *Getting Started with MongoDB* (page 52)
- *Generate Test Data* (page 57)

Administration

Replica Sets

- *Deploy a Replica Set* (page 607)

[93] http://api.mongodb.org/js/index.html
[94] https://github.com/mongodb/mongo/tree/master/jstests/
[95] https://jira.mongodb.org/browse/DOCS

- *Deploy Replica Set and Configure Authentication and Authorization* (page 348)
- *Convert a Standalone to a Replica Set* (page 619)
- *Add Members to a Replica Set* (page 620)
- *Remove Members from Replica Set* (page 622)
- *Replace a Replica Set Member* (page 624)
- *Adjust Priority for Replica Set Member* (page 625)
- *Resync a Member of a Replica Set* (page 640)
- *Deploy a Geographically Redundant Replica Set* (page 612)
- *Change the Size of the Oplog* (page 634)
- *Force a Member to Become Primary* (page 638)
- *Change Hostnames in a Replica Set* (page 649)
- *Add an Arbiter to Replica Set* (page 618)
- *Convert a Secondary to an Arbiter* (page 632)
- *Configure a Secondary's Sync Target* (page 652)
- *Configure a Delayed Replica Set Member* (page 629)
- *Configure a Hidden Replica Set Member* (page 628)
- *Configure Non-Voting Replica Set Member* (page 631)
- *Prevent Secondary from Becoming Primary* (page 626)
- *Configure Replica Set Tag Sets* (page 641)
- *Manage Chained Replication* (page 647)
- *Reconfigure a Replica Set with Unavailable Members* (page 645)
- *Recover Data after an Unexpected Shutdown* (page 274)
- *Troubleshoot Replica Sets* (page 654)

Sharding

- *Deploy a Sharded Cluster* (page 705)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 714)
- *Add Shards to a Cluster* (page 712)
- *Remove Shards from an Existing Sharded Cluster* (page 734)
- *Deploy Three Config Servers for Production Deployments* (page 713)
- *Migrate Config Servers with the Same Hostname* (page 722)
- *Migrate Config Servers with Different Hostnames* (page 723)
- *Replace Disabled Config Server* (page 724)
- *Migrate a Sharded Cluster to Different Hardware* (page 725)
- *Backup Cluster Metadata* (page 728)
- *Backup a Small Sharded Cluster with mongodump* (page 266)

- *Backup a Sharded Cluster with Filesystem Snapshots* (page 267)
- *Backup a Sharded Cluster with Database Dumps* (page 269)
- *Restore a Single Shard* (page 271)
- *Restore a Sharded Cluster* (page 272)
- *Schedule Backup Window for Sharded Clusters* (page 271)
- *Manage Shard Tags* (page 747)

Basic Operations

- *Use Database Commands* (page 234)
- *Recover Data after an Unexpected Shutdown* (page 274)
- *Expire Data from Collections by Setting TTL* (page 222)
- *Analyze Performance of Database Operations* (page 239)
- *Rotate Log Files* (page 243)
- *Build Old Style Indexes* (page 527)
- *Manage mongod Processes* (page 236)
- *Back Up and Restore with MongoDB Tools* (page 261)
- *Backup and Restore with Filesystem Snapshots* (page 256)

Security

- *Configure Linux iptables Firewall for MongoDB* (page 331)
- *Configure Windows netsh Firewall for MongoDB* (page 334)
- *Enable Client Access Control* (page 353)
- *Create a User Administrator* (page 381)
- *Add a User to a Database* (page 383)
- *Create a Role* (page 386)
- *Modify a User's Access* (page 391)
- *View Roles* (page 393)
- *Generate a Key File* (page 376)
- *Configure MongoDB with Kerberos Authentication on Linux* (page 369)
- *Create a Vulnerability Report* (page 402)

Development Patterns

- *Perform Two Phase Commits* (page 120)
- *Create an Auto-Incrementing Sequence Field* (page 130)
- *Enforce Unique Keys for Sharded Collections* (page 749)

- *Aggregation Examples* (page 453)
- *Model Data to Support Keyword Search* (page 172)
- *Limit Number of Elements in an Array after an Update* (page 114)
- *Perform Incremental Map-Reduce* (page 464)
- *Troubleshoot the Map Function* (page 466)
- *Troubleshoot the Reduce Function* (page 467)
- *Store a JavaScript Function on the Server* (page 247)

Text Search Patterns

- *Create a text Index* (page 543)
- *Specify a Language for Text Index* (page 544)
- *Specify Name for text Index* (page 546)
- *Control Search Results with Weights* (page 547)
- *Limit the Number of Entries Scanned* (page 548)

Data Modeling Patterns

- *Model One-to-One Relationships with Embedded Documents* (page 159)
- *Model One-to-Many Relationships with Embedded Documents* (page 160)
- *Model One-to-Many Relationships with Document References* (page 161)
- *Model Data for Atomic Operations* (page 171)
- *Model Tree Structures with Parent References* (page 164)
- *Model Tree Structures with Child References* (page 165)
- *Model Tree Structures with Materialized Paths* (page 168)
- *Model Tree Structures with Nested Sets* (page 170)

See also:

The MongoDB Manual contains administrative documentation and tutorials though out several sections. See *Replica Set Tutorials* (page 606) and *Sharded Cluster Tutorials* (page 704) for additional tutorials and information.

5.3 Administration Reference

UNIX ulimit Settings **(page 300)** Describes user resources limits (i.e. `ulimit`) and introduces the considerations and optimal configurations for systems that run MongoDB deployments.

System Collections **(page 304)** Introduces the internal collections that MongoDB uses to track per-database metadata, including indexes, collections, and authentication credentials.

Database Profiler Output **(page 305)** Describes the data collected by MongoDB's operation profiler, which introspects operations and reports data for analysis on performance and behavior.

Journaling Mechanics **(page 309)** Describes the internal operation of MongoDB's journaling facility and outlines how the journal allows MongoDB to provide provides durability and crash resiliency.

Exit Codes and Statuses **(page 311)** Lists the unique codes returned by `mongos` and `mongod` processes upon exit.

5.3.1 UNIX `ulimit` Settings

On this page

Most UNIX-like operating systems, including Linux and OS X, provide ways to limit and control the usage of system resources such as threads, files, and network connections on a per-process and per-user basis. These "ulimits" prevent single users from using too many system resources. Sometimes, these limits have low default values that can cause a number of issues in the course of normal MongoDB operation.

Note: Red Hat Enterprise Linux and CentOS 6 place a max process limitation of 1024 which overrides `ulimit` settings. Create a file named `/etc/security/limits.d/99-mongodb-nproc.conf` with new `soft nproc` and `hard nproc` values to increase the process limit. See `/etc/security/limits.d/90-nproc.conf` file as an example.

Resource Utilization

`mongod` and `mongos` each use threads and file descriptors to track connections and manage internal operations. This section outlines the general resource utilization patterns for MongoDB. Use these figures in combination with the actual information about your deployment and its use to determine ideal `ulimit` settings.

Generally, all `mongod` and `mongos` instances:

- track each incoming connection with a file descriptor *and* a thread.
- track each internal thread or *pthread* as a system process.

`mongod`

- 1 file descriptor for each data file in use by the `mongod` instance.
- 1 file descriptor for each journal file used by the `mongod` instance when `storage.journal.enabled` is `true`.
- In replica sets, each `mongod` maintains a connection to all other members of the set.

`mongod` uses background threads for a number of internal processes, including *TTL collections* (page 222), replication, and replica set health checks, which may require a small number of additional resources.

`mongos`

In addition to the threads and file descriptors for client connections, `mongos` must maintain connects to all config servers and all shards, which includes all members of all replica sets.

For `mongos`, consider the following behaviors:

- `mongos` instances maintain a connection pool to each shard so that the `mongos` can reuse connections and quickly fulfill requests without needing to create new connections.

- You can limit the number of incoming connections using the `maxIncomingConnections` run-time option. By restricting the number of incoming connections you can prevent a cascade effect where the `mongos` creates too many connections on the `mongod` instances.

  Note: Changed in version 2.6: MongoDB removed the upward limit on the `maxIncomingConnections` setting.

Review and Set Resource Limits

ulimit

You can use the `ulimit` command at the system prompt to check system limits, as in the following example:

```
$ ulimit -a
-t: cpu time (seconds)         unlimited
-f: file size (blocks)         unlimited
-d: data seg size (kbytes)     unlimited
-s: stack size (kbytes)        8192
-c: core file size (blocks)    0
-m: resident set size (kbytes) unlimited
-u: processes                  192276
-n: file descriptors           21000
-l: locked-in-memory size (kb) 40000
-v: address space (kb)         unlimited
-x: file locks                 unlimited
-i: pending signals            192276
-q: bytes in POSIX msg queues  819200
-e: max nice                   30
-r: max rt priority            65
-N 15:                         unlimited
```

`ulimit` refers to the per-*user* limitations for various resources. Therefore, if your `mongod` instance executes as a user that is also running multiple processes, or multiple `mongod` processes, you might see contention for these resources. Also, be aware that the `processes` value (i.e. `-u`) refers to the combined number of distinct processes and sub-process threads.

You can change `ulimit` settings by issuing a command in the following form:

```
ulimit -n <value>
```

There are both "hard" and the "soft" `ulimits` that affect MongoDB's performance. The "hard" `ulimit` refers to the maximum number of processes that a user can have active at any time. This is the ceiling: no non-root process can increase the "hard" `ulimit`. In contrast, the "soft" `ulimit` is the limit that is actually enforced for a session or process, but any process can increase it up to "hard" `ulimit` maximum.

A low "soft" `ulimit` can cause `can't create new thread, closing connection` errors if the number of connections grows too high. For this reason, it is extremely important to set *both* `ulimit` values to the recommended values.

`ulimit` will modify both "hard" and "soft" values unless the `-H` or `-S` modifiers are specified when modifying limit values.

For many distributions of Linux you can change values by substituting the `-n` option for any possible value in the output of `ulimit -a`. On OS X, use the `launchctl limit` command. See your operating system documentation for the precise procedure for changing system limits on running systems.

After changing the `ulimit` settings, you *must* restart the process to take advantage of the modified settings. You can use the `/proc` file system to see the current limitations on a running process.

Depending on your system's configuration, and default settings, any change to system limits made using `ulimit` may revert following system a system restart. Check your distribution and operating system documentation for more information.

Note: SUSE Linux Enterprise Server 11, and potentially other versions of SLES and other SUSE distributions, ship with virtual memory address space limited to 8GB by default. This *must* be adjusted in order to prevent virtual memory allocation failures as the database grows.

The SLES packages for MongoDB adjust these limits in the default scripts, but you will need to make this change manually if you are using custom scripts and/or the tarball release rather than the SLES packages.

Recommended `ulimit` Settings

Every deployment may have unique requirements and settings; however, the following thresholds and settings are particularly important for `mongod` and `mongos` deployments:

- `-f` (file size): `unlimited`
- `-t` (cpu time): `unlimited`
- `-v` (virtual memory): `unlimited` [96]
- `-n` (open files): `64000`
- `-m` (memory size): `unlimited` [1] [97]
- `-u` (processes/threads): `64000`

Always remember to restart your `mongod` and `mongos` instances after changing the `ulimit` settings to ensure that the changes take effect.

Linux distributions using Upstart

For Linux distributions that use Upstart, you can specify limits within service scripts if you start `mongod` and/or `mongos` instances as Upstart services. You can do this by using `limit` stanzas[98].

Specify the *Recommended ulimit Settings* (page 302), as in the following example:

```
limit fsize unlimited unlimited  # (file size)
limit cpu unlimited unlimited    # (cpu time)
limit as unlimited unlimited     # (virtual memory size)
limit nofile 64000 64000         # (open files)
limit nproc 64000 64000          # (processes/threads)
```

Each `limit` stanza sets the "soft" limit to the first value specified and the "hard" limit to the second.

After changing `limit` stanzas, ensure that the changes take effect by restarting the application services, using the following form:

```
restart <service name>
```

[96] If you limit virtual or resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

[97] The `-m` parameter to `ulimit` has no effect on Linux systems with kernel versions more recent than 2.4.30. You may omit `-m` if you wish.

[98] http://upstart.ubuntu.com/wiki/Stanzas#limit

Linux distributions using `systemd`

For Linux distributions that use `systemd`, you can specify limits within the `[Service]` sections of service scripts if you start `mongod` and/or `mongos` instances as `systemd` services. You can do this by using resource limit directives[99].

Specify the *Recommended ulimit Settings* (page 302), as in the following example:

```
[Service]
# Other directives omitted
# (file size)
LimitFSIZE=infinity
# (cpu time)
LimitCPU=infinity
# (virtual memory size)
LimitAS=infinity
# (open files)
LimitNOFILE=64000
# (processes/threads)
LimitNPROC=64000
```

Each `systemd` limit directive sets both the "hard" and "soft" limits to the value specified.

After changing `limit` stanzas, ensure that the changes take effect by restarting the application services, using the following form:

```
systemctl restart <service name>
```

`/proc` File System

Note: This section applies only to Linux operating systems.

The `/proc` file-system stores the per-process limits in the file system object located at `/proc/<pid>/limits`, where `<pid>` is the process's *PID* or process identifier. You can use the following `bash` function to return the content of the `limits` object for a process or processes with a given name:

```
return-limits(){

     for process in $@; do
          process_pids=`ps -C $process -o pid --no-headers | cut -d " " -f 2`

          if [ -z $@ ]; then
             echo "[no $process running]"
          else
             for pid in $process_pids; do
                   echo "[$process #$pid -- limits]"
                   cat /proc/$pid/limits
             done
          fi

     done

}
```

You can copy and paste this function into a current shell session or load it as part of a script. Call the function with one the following invocations:

[99] http://www.freedesktop.org/software/systemd/man/systemd.exec.html#LimitCPU=

```
return-limits mongod
return-limits mongos
return-limits mongod mongos
```

5.3.2 System Collections

On this page

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 664), specifically for replication purposes.

Collections

System collections include these collections stored in the `admin` database:

`admin.system.roles`
: New in version 2.6.

  The `admin.system.roles` (page 304) collection stores custom roles that administrators create and assign to users to provide access to specific resources.

`admin.system.users`
: Changed in version 2.6.

  The `admin.system.users` (page 304) collection stores the user's authentication credentials as well as any roles assigned to the user. Users may define authorization roles in the `admin.system.roles` (page 304) collection.

`admin.system.version`
: New in version 2.6.

  Stores the schema version of the user credential documents.

System collections also include these collections stored directly in each database:

`<database>.system.namespaces`
: The `<database>.system.namespaces` (page 304) 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 304) collection lists all the indexes in the database. Add and remove data from this collection via the `ensureIndex()` and `dropIndex()`

`<database>.system.profile`
: The `<database>.system.profile` (page 304) collection stores database profiling information. For information on profiling, see *Database Profiling* (page 230).

`<database>.system.js`
: The `<database>.system.js` (page 304) collection holds special JavaScript code for use in *server side JavaScript* (page 277). See *Store a JavaScript Function on the Server* (page 247) for more information.

5.3.3 Database Profiler Output

On this page

The database profiler captures data information about read and write operations, cursor operations, and database commands. To configure the database profile and set the thresholds for capturing profile data, see the *Analyze Performance of Database Operations* (page 239) section.

The database profiler writes data in the `system.profile` (page 304) collection, which is a *capped collection*. To view the profiler's output, use normal MongoDB queries on the `system.profile` (page 304) collection.

Note: Because the database profiler writes data to the `system.profile` (page 304) collection in a database, the profiler will profile some write activity, even for databases that are otherwise read-only.

Example `system.profile` Document

The documents in the `system.profile` (page 304) collection have the following form. This example document reflects an update operation:

```
{
    "op" : "update",
    "ns" : "social.users",
    "query" : {
        "name" : "j.r."
    },
    "updateobj" : {
        "$set" : {
            "likes" : [
                "basketball",
                "trekking"
            ]
        }
    },
    "nscanned" : 1,
    "nscannedObjects" : 1,
    "moved" : true,
    "nmoved" : 1,
    "nMatched" : 1,
    "nModified" : 1,
    "keyUpdates" : 0,
    "numYield" : 0,
    "lockStats" : {
        "timeLockedMicros" : {
            "r" : NumberLong(0),
            "w" : NumberLong(258)
        },
        "timeAcquiringMicros" : {
```

```
            "r" : NumberLong(0),
            "w" : NumberLong(7)
        }
    },
    "millis" : 0,
    "execStats" : {
    },
    "ts" : ISODate("2012-12-10T19:31:28.977Z"),
    "client" : "127.0.0.1",
    "allUsers" : [ ],
    "user" : ""
}
```

Output Reference

For any single operation, the documents created by the database profiler will include a subset of the following fields. The precise selection of fields in these documents depends on the type of operation.

`system.profile.`**`op`**
The type of operation. The possible values are:

- `insert`
- `query`
- `update`
- `remove`
- `getmore`
- `command`

`system.profile.`**`ns`**
The *namespace* the operation targets. Namespaces in MongoDB take the form of the *database*, followed by a dot (`.`), followed by the name of the *collection*.

`system.profile.`**`query`**
The *query document* (page 100) used.

Changed in version 2.6.11: For `"getmore"` (page 306) operations on cursors returned from a `db.collection.find()` or a `db.collection.aggregate()`, the `query` (page 306) field contains respectively the query predicate or the issued `aggregate` command document. For details on the `aggregate` command document, see the `aggregate` reference page.

`system.profile.`**`command`**
The command operation.

`system.profile.`**`updateobj`**
The `<update>` document passed in during an *update* (page 107) operation.

`system.profile.`**`cursorid`**
The ID of the cursor accessed by a `getmore` operation.

`system.profile.`**`ntoreturn`**
Changed in version 2.2: In 2.0, MongoDB includes this field for `query` and `command` operations. In 2.2, this information MongoDB also includes this field for `getmore` operations.

The number of documents the operation specified to return. For example, the `profile` command would return one document (a results document) so the `ntoreturn` (page 306) value would be `1`. The `limit(5)` command would return five documents so the `ntoreturn` (page 306) value would be `5`.

If the ntoreturn (page 306) value is 0, the command did not specify a number of documents to return, as would be the case with a simple find() command with no limit specified.

system.profile.**ntoskip**
New in version 2.2.

The number of documents the skip() method specified to skip.

system.profile.**nscanned**
The number of documents that MongoDB scans in the *index* (page 481) in order to carry out the operation.

In general, if nscanned (page 307) is much higher than nreturned (page 308), the database is scanning many objects to find the target objects. Consider creating an index to improve this.

system.profile.**nscannedObjects**
The number of documents that MongoDB scans from the collection in order to carry out the operation.

system.profile.**moved**
This field appears with a value of true when an update operation moved one or more documents to a new location on disk. If the operation did not result in a move, this field does not appear. Operations that result in a move take more time than in-place updates and typically occur as a result of document growth.

system.profile.**nmoved**
New in version 2.2.

The number of documents the operation moved on disk. This field appears only if the operation resulted in a move. The field's implicit value is zero, and the field is present only when non-zero.

system.profile.**scanAndOrder**
scanAndOrder (page 307) is a boolean that is true when a query **cannot** use the order of documents in the index for returning sorted results: MongoDB must sort the documents after it receives the documents from a cursor.

If scanAndOrder (page 307) is false, MongoDB *can* use the order of the documents in an index to return sorted results.

system.profile.**ndeleted**
The number of documents deleted by the operation.

system.profile.**ninserted**
The number of documents inserted by the operation.

system.profile.**nMatched**
New in version 2.6.

The number of documents that match the system.profile.query (page 306) condition for the update operation.

system.profile.**nModified**
New in version 2.6.

The number of documents modified by the update operation.

system.profile.**upsert**
A boolean that indicates the update operation's upsert option value. Only appears if upsert is true.

system.profile.**keyUpdates**
New in version 2.2.

The number of *index* (page 481) 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 778).

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*, this value may be longer than the total length of the operation (i.e. `millis` (page 308).)

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 308) can affect performance. To limit the size of the result document for a query operation, you can use any of the following:

- *Projections* (page 112)
- `The limit() method`
- `The batchSize() method`

Note: When MongoDB writes query profile information to the log, the `responseLength` (page 308) value is in a field named `reslen`.

---

system.profile.**millis**
The time in milliseconds from the perspective of the `mongod` from the beginning of the operation to the end of the operation.

system.profile.**execStats**
New in version 2.6.

A document that contains the execution statistics of the query operation. For other operations, the value is an empty document.

The `system.profile.execStats` (page 308) presents the staticstics as a tree; each node provides the statistics for the operation executed during that stage of the query operation.

Note: The following fields list for `execStats` (page 308) is not meant to be exhaustive as the returned fields vary per stage.

`system.profile.execStats.type`
The descriptive name for the operation performed as part of the query execution; e.g.

- `COLLSCAN` for a collection scan
- `IXSCAN` for scanning index keys
- `FETCH` for retrieving documents

`system.profile.execStats.children`
An array that contains statistics for the operations that are the children of the current stage.

`system.profile.ts`
The timestamp of the operation.

`system.profile.client`
The IP address or hostname of the client connection where the operation originates.

For some operations, such as `db.eval()`, the client is `0.0.0.0:0` instead of an actual client.

`system.profile.allUsers`
An array of authenticated user information (user name and database) for the session. See also *Client Authentication* (page 318).

`system.profile.user`
The authenticated user who ran the operation. If the operation was not run by an authenticated user, this field's value is an empty string.

5.3.4 Journaling Mechanics

On this page

When running with journaling, MongoDB stores and applies *write operations* (page 77) in memory and in the on-disk journal before the changes are present in the data files on disk. Writes to the journal are atomic, ensuring the consistency of the on-disk journal files. This document discusses the implementation and mechanics of journaling in MongoDB systems. See *Manage Journaling* (page 245) for information on configuring, tuning, and managing journaling.

Journal Files

With journaling enabled, MongoDB creates a journal subdirectory within the directory defined by `dbPath`, which is `/data/db` by default. The journal directory holds journal files, which contain write-ahead redo logs. The directory also holds a last-sequence-number file. A clean shutdown removes all the files in the journal directory. A dirty shutdown (crash) leaves files in the journal directory; these are used to automatically recover the database to a consistent state when the mongod process is restarted.

Journal 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 a particular journal file to the database data files, it deletes the file, as it is no longer needed for recovery purposes. Unless you write *many* bytes of data per second, the journal directory should contain only two or three journal files.

You can use the `storage.smallFiles` run time option when starting `mongod` to limit the size of each journal file to 128 megabytes, if you prefer.

To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different filesystem from the database data files.

Important: If you place the journal on a different filesystem from your data files you *cannot* use a filesystem snapshot alone to capture valid backups of a `dbPath` directory. In this case, use `fsyncLock()` to ensure that database files are consistent before the snapshot and `fsyncUnlock()` once the snapshot is complete.

Note: Depending on your filesystem, you might experience a preallocation lag the first time you start a `mongod` instance with journaling enabled.

MongoDB may preallocate journal files if the `mongod` process determines that it is more efficient to preallocate journal files than create new journal files as needed. The amount of time required to pre-allocate lag might last several minutes, during which you will not be able to connect to the database. This is a one-time preallocation and does not occur with future invocations.

To avoid preallocation lag, see *Avoid Preallocation Lag* (page 245).

Storage Views used in Journaling

With journaling, MongoDB's storage layer has two internal views of the data set.

The `shared view` stores modified data for upload to the MongoDB data files. The `shared view` is the only view with direct access to the MongoDB data files. When running with journaling, `mongod` asks the operating system to map your existing on-disk data files to the `shared view` virtual memory view. The operating system maps the files but does not load them. MongoDB later loads data files into the `shared view` as needed.

The `private view` stores data for use with *read operations* (page 64). The `private view` is the first place MongoDB applies new *write operations* (page 77). Upon a journal commit, MongoDB copies the changes made in the `private view` to the `shared view`, where they are then available for uploading to the database data files.

The journal is an on-disk view that stores new write operations after MongoDB applies the operation to the `private view` but before applying them to the data files. The journal provides durability. If the `mongod` instance were to crash without having applied the writes to the data files, the journal could replay the writes to the `shared view` for eventual upload to the data files.

How Journaling Records Write Operations

MongoDB copies the write operations to the journal in batches called group commits. These "group commits" help minimize the performance impact of journaling, since a group commit must block all writers during the commit. See `commitIntervalMs` for information on the default commit interval.

Journaling stores raw operations that allow MongoDB to reconstruct the following:

- document insertion/updates
- index modifications
- metadata changes to the namespace files
- creation and dropping of databases and their associated data files

As *write operations* (page 77) 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. Each journal entry describes the 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. The operating system may choose to flush the `shared view` to disk at a higher frequency than 60 seconds, particularly if the system is low on free memory.

When MongoDB flushes write operations to the data files, MongoDB notes which journal writes have been flushed. Once a journal file contains only flushed writes, it is no longer needed for recovery, and MongoDB either deletes it or recycles it for a new journal file.

As part of journaling, MongoDB routinely asks the operating system to remap the `shared view` to the `private view`, in order to save physical RAM. Upon a new remapping, the operating system knows that physical memory pages can be shared between the `shared view` and the `private view` mappings.

Note: The interaction between the `shared view` and the on-disk data files is similar to how MongoDB works *without* journaling, which is that MongoDB asks the operating system to flush in-memory changes back to the data files every 60 seconds.

5.3.5 Exit Codes and Statuses

MongoDB will return one of the following codes and statuses when exiting. Use this guide to interpret logs and when troubleshooting issues with `mongod` and `mongos` instances.

0

Returned by MongoDB applications upon successful exit.

2

The specified options are in error or are incompatible with other options.

3

Returned by `mongod` if there is a mismatch between hostnames specified on the command line and in the `local.sources` (page 667) collection. `mongod` may also return this status if *oplog* collection in the `local` database is not readable.

4

The version of the database is different from the version supported by the `mongod` (or `mongod.exe`) instance. The instance exits cleanly. Restart `mongod` with the `--upgrade` option to upgrade the database to the version supported by this `mongod` instance.

5

Returned by `mongod` if a `moveChunk` operation fails to confirm a commit.

12

Returned by the `mongod.exe` process on Windows when it receives a Control-C, Close, Break or Shutdown event.

14

Returned by MongoDB applications which encounter an unrecoverable error, an uncaught exception or uncaught signal. The system exits without performing a clean shut down.

20

Message: `ERROR: wsastartup failed <reason>`

Returned by MongoDB applications on Windows following an error in the WSAStartup function.

Message: `NT Service Error`

Returned by MongoDB applications for Windows due to failures installing, starting or removing the NT Service for the application.

45

Returned when a MongoDB application cannot open a file or cannot obtain a lock on a file.

47

MongoDB applications exit cleanly following a large clock skew (32768 milliseconds) event.

48

`mongod` exits cleanly if the server socket closes. The server socket is on port `27017` by default, or as specified to the `--port` run-time option.

49

Returned by `mongod.exe` or `mongos.exe` on Windows when either receives a shutdown message from the *Windows Service Control Manager*.

100

Returned by `mongod` when the process throws an uncaught exception.

CHAPTER 6

Security

This section outlines basic security and risk management strategies and access control. The included tutorials outline specific tasks for configuring firewalls, authentication, and system privileges.

Security Introduction **(page 313)** A high-level introduction to security and MongoDB deployments.

Security Concepts **(page 316)** The core documentation of security.

> ***Authentication*** **(page 316)** Mechanisms for verifying user and instance access to MongoDB.
>
> ***Authorization*** **(page 320)** Control access to MongoDB instances using authorization.
>
> ***Network Exposure and Security*** **(page 322)** Discusses potential security risks related to the network and strategies for decreasing possible network-based attack vectors for MongoDB.
>
> Continue reading from *Security Concepts* (page 316) for additional documentation of MongoDB's security features and operation.

Security Tutorials **(page 329)** Tutorials for enabling and configuring security features for MongoDB.

> ***Network Security Tutorials*** **(page 330)** Ensure that the underlying network configuration supports a secure operating environment for MongoDB deployments, and appropriately limits access to MongoDB deployments.
>
> ***Access Control Tutorials*** **(page 352)** These tutorials describe procedures relevant for the configuration, operation, and maintenance of MongoDB's access control system.
>
> ***User and Role Management Tutorials*** **(page 381)** MongoDB's access control system provides a flexible role-based access control system that you can use to limit access to MongoDB deployments. The tutorials in this section describe the configuration an setup of the authorization system.
>
> Continue reading from *Security Tutorials* (page 329) for additional tutorials that address the use and management of secure MongoDB deployments.

Create a Vulnerability Report **(page 402)** Report a vulnerability in MongoDB.

Security Reference **(page 403)** Reference for security related functions.

Security Checklist **(page 431)** A high level overview of global security consideration for administrators of MongoDB deployments. Use this checklist if you are new to deploying MongoDB in production and want to implement high quality security practices.

6.1 Security Introduction

On this page

Maintaining a secure MongoDB deployment requires administrators to implement controls to ensure that users and applications have access to only the data that they require. MongoDB provides features that allow administrators to implement these controls and restrictions for any MongoDB deployment.

If you are already familiar with security and MongoDB security practices, consider the *Security Checklist* (page 431) for a collection of recommended actions to protect a MongoDB deployment.

6.1.1 Authentication

Before gaining access to a system all clients should identify themselves to MongoDB. This ensures that no client can access the data stored in MongoDB without being explicitly allowed.

MongoDB supports a number of *authentication mechanisms* (page 317) that clients can use to verify their identity. MongoDB supports two mechanisms: a password-based challenge and response protocol and x.509 certificates. Additionally, MongoDB Enterprise[1] also provides support for *LDAP proxy authentication* (page 318) and *Kerberos authentication* (page 318).

See *Authentication* (page 316) for more information.

6.1.2 Role Based Access Control

Access control, i.e. *authorization* (page 320), determines a user's access to resources and operations. Clients should only be able to perform the operations required to fulfill their approved functions. This is the "principle of least privilege" and limits the potential risk of a compromised application.

MongoDB's role-based access control system allows administrators to control all access and ensure that all granted access applies as narrowly as possible. MongoDB does not enable authorization by default. When you enable *authorization* (page 320), MongoDB will require authentication for all connections.

When authorization is enabled, MongoDB controls a user's access through the roles assigned to the user. A role consists of a set of privileges, where a privilege consists of *actions*, or a set of operations, and a *resource* upon which the actions are allowed.

Users may have one or more role that describes their access. MongoDB provides several *built-in roles* (page 405) and users can construct specific roles tailored to clients' actual requirements.

See *Authorization* (page 320) for more information.

6.1.3 Auditing

Auditing provides administrators with the ability to verify that the implemented security policies are controlling activity in the system. Retaining audit information ensures that administrators have enough information to perform forensic investigations and comply with regulations and polices that require audit data.

[1] http://www.mongodb.com/products/mongodb-enterprise

See *Auditing* (page 325) for more information.

6.1.4 Encryption

Transport Encryption

You can use TLS/SSL (Transport Layer Security/Secure Sockets Layer) to encrypt all of MongoDB's network traffic. TLS/SSL ensures that MongoDB network traffic is only readable by the intended client.

See *Configure mongod and mongos for TLS/SSL* (page 338) for more information.

Encryption at Rest

There are two broad classes of approaches to encrypting data at rest with MongoDB. You can use these solutions together or independently:

Application Level Encryption

Provide encryption on a per-field or per-document basis within the application layer. To encrypt document or field level data, write custom encryption and decryption routines or use a commercial solutions such as the Vormetric Data Security Platform[2].

Storage Encryption

Encrypt all MongoDB data on the storage or operating system to ensure that only authorized processes can access protected data. A number of third-party libraries can integrate with the operating system to provide transparent disk-level encryption. For example:

Linux Unified Key Setup (LUKS) LUKS is available for most Linux distributions. For configuration explanation, see the LUKS documentation from Red Hat[3].

IBM Guardium Data Encryption IBM Guardium Data Encryption[4] provides support for disk-level encryption for Linux and Windows operating systems.

Vormetric Data Security Platform The Vormetric Data Security Platform[5] provides disk and file-level encryption in addition to application level encryption.

Bitlocker Drive Encryption Bitlocker Drive Encryption[6] is a feature available on Windows Server 2008 and 2012 that provides disk encryption.

Properly configured disk encryption, when used alongside good security policies that protect relevant accounts, passwords, and encryption keys, can help ensure compliance with standards, including HIPAA, PCI-DSS, and FERPA.

6.1.5 Hardening Deployments and Environments

In addition to implementing controls within MongoDB, you should also place controls around MongoDB to reduce the risk exposure of the entire MongoDB system. This is a *defense in depth* strategy.

[2] http://www.vormetric.com/sites/default/files/sb-MongoDB-Letter-2014-0611.pdf
[3] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Security_Guide/sec-Encryption.html
[4] http://www-03.ibm.com/software/products/en/infosphere-guardium-data-encryption
[5] http://www.vormetric.com/sites/default/files/sb-MongoDB-Letter-2014-0611.pdf
[6] http://technet.microsoft.com/en-us/library/hh831713.aspx

Hardening MongoDB extends the ideas of least privilege, auditing, and encryption outside of MongoDB. Reducing risk includes: configuring the network rules to ensure that only trusted hosts have access to MongoDB, and that the MongoDB processes only have access to the parts of the filesystem required for operation.

6.1.6 Additional Resources

- Making HIPAA Compliant MongoDB Applications[7]
- Security Architecture White Paper[8]
- Webinar: Securing Your MongoDB Deployment[9]

6.2 Security Concepts

These documents introduce and address concepts and strategies related to security practices in MongoDB deployments.

Authentication **(page 316)** Mechanisms for verifying user and instance access to MongoDB.

Authorization **(page 320)** Control access to MongoDB instances using authorization.

Collection-Level Access Control **(page 322)** Scope privileges to specific collections.

Network Exposure and Security **(page 322)** Discusses potential security risks related to the network and strategies for decreasing possible network-based attack vectors for MongoDB.

Security and MongoDB API Interfaces **(page 324)** Discusses potential risks related to MongoDB's JavaScript, HTTP and REST interfaces, including strategies to control those risks.

Auditing **(page 325)** Audit server and client activity for `mongod` and `mongos` instances.

Kerberos Authentication **(page 326)** Kerberos authentication and MongoDB.

6.2.1 Authentication

On this page

Authentication is the process of verifying the identity of a client. When access control, i.e. *authorization* (page 320), is enabled, MongoDB requires all clients to authenticate themselves first in order to determine the access for the client.

Although authentication and *authorization* (page 320) are closely connected, authentication is distinct from authorization. Authentication verifies the identity of a user; authorization determines the verified user's access to resources and operations.

MongoDB supports a number of *authentication mechanisms* (page 317) that clients can use to verify their identity. These mechanisms allow MongoDB to integrate into your existing authentication system. See *Authentication Mechanisms* (page 317) for details.

[7] https://www.mongodb.com/blog/post/making-hipaa-compliant-applications-mongodb?jmp=docs
[8] https://www.mongodb.com/lp/white-paper/mongodb-security-architecture?jmp=docs
[9] http://www.mongodb.com/presentations/webinar-securing-your-mongodb-deployment?jmp=docs

In addition to verifying the identity of a client, MongoDB can require members of replica sets and sharded clusters to *authenticate their membership* (page 318) to their respective replica set or sharded cluster. See *Authentication Between MongoDB Instances* (page 318) for more information.

Client Users

To authenticate a client in MongoDB, you must add a corresponding user to MongoDB. When adding a user, you create the user in a specific database. Together, the user's name and database serve as a unique identifier for that user. That is, if two users have the same name but are created in different databases, they are two separate users. To authenticate, the client must authenticate the user against the user's database. For instance, if using the `mongo` shell as a client, you can specify the database for the user with the *–authenticationDatabase* option.

To add and manage user information, MongoDB provides the `db.createUser()` method as well as other *user management methods*. For an example of adding a user to MongoDB, see *Add a User to a Database* (page 383).

MongoDB stores all user information, including `name` (page 416), `password` (page 416), and the `user's database` (page 416), in the *system.users* (page 415) collection in the `admin` database.

Authentication Mechanisms

MongoDB supports multiple authentication mechanisms. MongoDB's default authentication method is a *challenge and response mechanism (MONGODB-CR)* (page 317). MongoDB also supports *x509 certificate authentication* (page 317), *LDAP proxy authentication* (page 318), and *Kerberos authentication* (page 318).

This section introduces the mechanisms available in MongoDB.

To specify the authentication mechanism to use, see `authenticationMechanisms`.

MONGODB-CR Authentication

`MONGODB-CR` is a challenge-response mechanism that authenticates users through passwords. `MONGODB-CR` is the default mechanism.

When you use `MONGODB-CR` authentication, `MONGODB-CR` verifies the user against the user's `name` (page 416), `password` (page 416) and `database` (page 416). The user's database is the database where the user was created, and the user's database and the user's name together serves to identify the user.

Using `key files`, you can also use `MONGODB-CR` authentication for the *internal member authentication* (page 318) of replica set members and sharded cluster members. The contents of the key files serve as the shared password for the members. You must store the key file on each `mongod` or `mongos` instance for that replica set or sharded cluster. The content of the key file is arbitrary but must be the same on all `mongod` and `mongos` instances that connect to each other.

See *Generate a Key File* (page 376) for instructions on generating a key file and turning on key file authentication for members.

x.509 Certificate Authentication

New in version 2.6.

MongoDB supports x.509 certificate authentication for use with a secure *TLS/SSL connection* (page 338).

To authenticate to servers, clients can use x.509 certificates instead of usernames and passwords. See *Client x.509 Certificate* (page 357) for more information.

For membership authentication, members of sharded clusters and replica sets can use x.509 certificates instead of key files. See *Use x.509 Certificate for Membership Authentication* (page 359) for more information.

Kerberos Authentication

MongoDB Enterprise[10] supports authentication using a Kerberos service. Kerberos is an industry standard authentication protocol for large client/server systems.

To use MongoDB with Kerberos, you must have a properly configured Kerberos deployment, configured *Kerberos service principals* (page 327) for MongoDB, and added *Kerberos user principal* (page 327) to MongoDB.

See *Kerberos Authentication* (page 326) for more information on Kerberos and MongoDB. To configure MongoDB to use Kerberos authentication, see *Configure MongoDB with Kerberos Authentication on Linux* (page 369) and *Configure MongoDB with Kerberos Authentication on Windows* (page 372).

LDAP Proxy Authority Authentication

MongoDB Enterprise[11] supports proxy authentication through a Lightweight Directory Access Protocol (LDAP) service. See *Authenticate Using SASL and LDAP with OpenLDAP* (page 366) and *Authenticate Using SASL and LDAP with ActiveDirectory* (page 363).

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB does **not** support LDAP authentication in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards.

Authentication Behavior

Client Authentication

Clients can authenticate using the *challenge and response* (page 317), *x.509* (page 317), *LDAP Proxy* (page 318) and *Kerberos* (page 318) mechanisms.

Each client connection should authenticate as exactly one user. If a client authenticates to a database as one user and later authenticates to the same database as a different user, the second authentication invalidates the first. While clients can authenticate as multiple users if the users are defined on different databases, we recommend authenticating as one user at a time, providing the user with appropriate privileges on the databases required by the user.

See *Authenticate to a MongoDB Instance or Cluster* (page 375) for more information.

Authentication Between MongoDB Instances

You can authenticate members of *replica sets* and *sharded clusters*. To authenticate members of a single MongoDB deployment to each other, MongoDB can use the `keyFile` and *x.509* (page 317) mechanisms. Using `keyFile` authentication for members also enables authorization.

Always run replica sets and sharded clusters in a trusted networking environment. Ensure that the network permits only trusted traffic to reach each `mongod` and `mongos` instance.

[10] http://www.mongodb.com/products/mongodb-enterprise
[11] http://www.mongodb.com/products/mongodb-enterprise

Use your environment's firewall and network routing to ensure that traffic *only* from clients and other members can reach your `mongod` and `mongos` instances. If needed, use virtual private networks (VPNs) to ensure secure connections over wide area networks (WANs).

Always ensure that:

- Your network configuration will allow every member of the replica set or sharded cluster to contact every other member.
- If you use MongoDB's authentication system to limit access to your infrastructure, ensure that you configure a `keyFile` on all members to permit authentication.

See *Generate a Key File* (page 376) for instructions on generating a key file and turning on key file authentication for members. For an example of using key files for sharded cluster authentication, see *Enable Authentication in a Sharded Cluster* (page 354).

Authentication on Sharded Clusters

In sharded clusters, applications authenticate to directly to `mongos` instances, using credentials stored in the `admin` database of the *config servers*. The shards in the sharded cluster also have credentials, and clients can authenticate directly to the shards to perform maintenance directly on the shards. In general, applications and clients should connect to the sharded cluster through the `mongos`.

Changed in version 2.6: Previously, the credentials for authenticating to a database on a cluster resided on the *primary shard* (page 683) for that database.

Some maintenance operations, such as `cleanupOrphaned`, `compact`, `rs.reconfig()`, require direct connections to specific shards in a sharded cluster. To perform these operations with authentication enabled, you must connect directly to the shard and authenticate as a *shard local* administrative user. To create a *shard local* administrative user, connect directly to the shard and create the user. MongoDB stores *shard local* users in the `admin` database of the shard itself. These *shard local* users are completely independent from the users added to the sharded cluster via `mongos`. *Shard local* users are local to the shard and are inaccessible by `mongos`. Direct connections to a shard should only be for shard-specific maintenance and configuration.

Localhost Exception

The localhost exception allows you to enable authorization before creating the first user in the system. When active, the localhost exception allows all connections from the localhost interface to have full access to that instance. The exception applies only when there are no users created in the MongoDB instance.

If you use the localhost exception when deploying a new MongoDB system, the first user you create must be in the `admin` database with privileges to create other users, such as a user with the `userAdmin` (page 407) or `userAdminAnyDatabase` (page 411) role. See *Enable Client Access Control* (page 353) and *Create a User Administrator* (page 381) for more information.

In the case of a sharded cluster, the localhost exception can apply to the cluster as a whole or separately to each shard. The localhost exception can apply to the cluster as a whole if there are no user information stored on the config servers *and* clients access via `mongos` instances.

The localhost exception can apply separately to each shard if there is no user information stored on the shard itself and clients connect to the shard directly.

To prevent unauthorized access to a cluster's shards, you must either create an administrator on each shard or disable the localhost exception. To disable the localhost exception, use `setParameter` to set the `enableLocalhostAuthBypass` parameter to `0` during startup.

6.2.2 Authorization

On this page

MongoDB employs Role-Based Access Control (RBAC) to govern access to a MongoDB system. A user is granted one or more *roles* (page 320) that determine the user's access to database resources and operations. Outside of role assignments, the user has no access to the system.

MongoDB does not enable authorization by default. You can enable authorization using the `--auth` or the `--keyFile` options, or if using a configuration file, with the `security.authorization` or the `security.keyFile` settings.

MongoDB provides *built-in roles* (page 405), each with a dedicated purpose for a common use case. Examples include the `read` (page 405), `readWrite` (page 405), `dbAdmin` (page 406), and `root` (page 412) roles.

Administrators also can create new roles and privileges to cater to operational needs. Administrators can assign privileges scoped as granularly as the collection level.

When granted a role, a user receives all the privileges of that role. A user can have several roles concurrently, in which case the user receives the union of all the privileges of the respective roles.

Roles

A role consists of privileges that pair resources with allowed operations. Each privilege is specified explicitly in the role or inherited from another role or both.

Except for roles created in the `admin` database, a role can only include privileges that apply to its database and can only inherit from other roles in its database.

A role created in the `admin` database can include privileges that apply to the `admin` database, other databases or to the *cluster* (page 418) resource, and can inherit from roles in other databases as well as the `admin` database.

A user assigned a role receives all the privileges of that role. The user can have multiple roles and can have different roles on different databases.

Roles always grant privileges and never limit access. For example, if a user has both `read` (page 405) *and* `readWriteAnyDatabase` (page 411) roles on a database, the greater access prevails.

Privileges

A privilege consists of a specified resource and the actions permitted on the resource.

A privilege *resource* (page 417) is either a database, collection, set of collections, or the cluster. If the cluster, the affiliated actions affect the state of the system rather than a specific database or collection.

An *action* (page 418) is a command or method the user is allowed to perform on the resource. A resource can have multiple allowed actions. For available actions see *Privilege Actions* (page 418).

For example, a privilege that includes the `update` (page 419) action allows a user to modify existing documents on the resource. To additionally grant the user permission to create documents on the resource, the administrator would add the `insert` (page 419) action to the privilege.

For privilege syntax, see `admin.system.roles.privileges` (page 413).

Inherited Privileges

A role can include one or more existing roles in its definition, in which case the role inherits all the privileges of the included roles.

A role can inherit privileges from other roles in its database. A role created on the `admin` database can inherit privileges from roles in any database.

User-Defined Roles

New in version 2.6.

User administrators can create custom roles to ensure collection-level and command-level granularity and to adhere to the policy of *least privilege*. Administrators create and edit roles using the *role management commands*.

MongoDB scopes a user-defined role to the database in which it is created and uniquely identifies the role by the pairing of its name and its database. MongoDB stores the roles in the `admin` database's *system.roles* (page 412) collection. Do not access this collection directly but instead use the *role management commands* to view and edit custom roles.

Collection-Level Access Control

By creating a role with *privileges* (page 320) that are scoped to a specific collection in a particular database, administrators can implement collection-level access control.

See *Collection-Level Access Control* (page 322) for more information.

Users

MongoDB stores user credentials in the protected `admin.system.users` (page 304). Use the *user management methods* to view and edit user credentials.

Role Assignment to Users

User administrators create the users that access the system's databases. MongoDB's *user management commands* let administrators create users and assign them roles.

MongoDB scopes a user to the database in which the user is created. MongoDB stores all user definitions in the `admin` database, no matter which database the user is scoped to. MongoDB stores users in the `admin` database's *system.users collection* (page 415). Do not access this collection directly but instead use the *user management commands*.

The first role assigned in a database should be either `userAdmin` (page 407) or `userAdminAnyDatabase` (page 411). This user can then create all other users in the system. See *Create a User Administrator* (page 381).

Protect the User and Role Collections

MongoDB stores role and user data in the protected `admin.system.roles` (page 304) and `admin.system.users` (page 304) collections, which are only accessible using the *user management methods*.

If you disable access control, **do not** modify the `admin.system.roles` (page 304) and `admin.system.users` (page 304) collections using normal `insert()` and `update()` operations.

Additional Information

See the reference section for documentation of all *built-in-roles* (page 405) and all available *privilege actions* (page 418). Also consider the reference for the form of the *resource documents* (page 417).

To create users see the *Create a User Administrator* (page 381) and *Add a User to a Database* (page 383) tutorials.

6.2.3 Collection-Level Access Control

On this page

Collection-level access control allows administrators to grant users privileges that are scoped to specific collections.

Administrators can implement collection-level access control through *user-defined roles* (page 321). By creating a role with *privileges* (page 320) that are scoped to a specific collection in a particular database, administrators can provision users with roles that grant privileges on a collection level.

Privileges and Scope

A privilege consists of *actions* (page 418) and the *resources* (page 417) upon which the actions are permissible; i.e. the resources define the scope of the actions for that privilege.

By specifying both the database and the collection in the *resource document* (page 417) for a privilege, administrator can limit the privilege actions just to a specific collection in a specific database. Each privilege action in a role can be scoped to a different collection.

For example, a user defined role can contain the following privileges:

```
privileges: [
  { resource: { db: "products", collection: "inventory" }, actions: [ "find", "update", "insert" ] },
  { resource: { db: "products", collection: "orders" },  actions: [ "find" ] }
]
```

The first privilege scopes its actions to the `inventory` collection of the `products` database. The second privilege scopes its actions to the `orders` collection of the `products` database.

Additional Information

For more information on user-defined roles and MongoDB authorization model, see *Authorization* (page 320). For a tutorial on creating user-defined roles, see *Create a Role* (page 386).

6.2.4 Network Exposure and Security

On this page

By default, MongoDB programs (i.e. `mongos` and `mongod`) will bind to all available network interfaces (i.e. IP addresses) on a system.

This page outlines various runtime options that allow you to limit access to MongoDB programs.

Configuration Options

You can limit the network exposure with the following `mongod` and `mongos` configuration options: `enabled`, `net.http.RESTInterfaceEnabled`, `bindIp`, and `port`. You can use a `configuration file` to specify these settings.

nohttpinterface

The `enabled` setting for `mongod` and `mongos` instances disables the "home" status page.

Changed in version 2.6: The `mongod` and `mongos` instances run with the http interface *disabled* by default.

The status interface is read-only by default, and the default port for the status page is `28017`. Authentication does not control or affect access to this interface.

Important: Disable this interface for production deployments. If you *enable* this interface, you should only allow trusted clients to access this port. See *Firewalls* (page 324).

---

rest

The `net.http.RESTInterfaceEnabled` setting for `mongod` enables a fully interactive administrative *REST* interface, which is *disabled* by default. The `net.http.RESTInterfaceEnabled` configuration makes the http status interface [12], which is read-only by default, fully interactive. Use the `net.http.RESTInterfaceEnabled` setting with the `enabled` setting.

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 --httpinterface`*.

Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

---

bind_ip

The `bindIp` setting for `mongod` and `mongos` instances limits the network interfaces on which MongoDB programs will listen for incoming connections. You can also specify a number of interfaces by passing `bindIp` a comma separated list of IP addresses. You can use the *`mongod --bind_ip`* and *`mongos --bind_ip`* option on the command line at run time to limit the network accessibility of a MongoDB program.

Important: Make sure that your `mongod` and `mongos` instances are only accessible on trusted networks. If your system has more than one network interface, bind MongoDB programs to the private or internal network interface.

---

[12] Starting in version 2.6, http interface is *disabled* by default.

port

The `port` setting for `mongod` and `mongos` instances changes the main port on which the `mongod` or `mongos` instance listens for connections. The default port is `27017`. Changing the port does not meaningfully reduce risk or limit exposure. You may also specify this option on the command line as *`mongod --port`* or *`mongos --port`*. Setting `port` also indirectly sets the port for the HTTP status interface, which is always available on the port numbered `1000` greater than the primary `mongod` port.

Only allow trusted clients to connect to the port for the `mongod` and `mongos` instances. See *Firewalls* (page 324).

See also *Security Considerations* (page 204) and *Default MongoDB Port* (page 424).

Firewalls

Firewalls allow administrators to filter and control access to a system by providing granular control over what network communications. For administrators of MongoDB, the following capabilities are important: limiting incoming traffic on a specific port to specific systems, and limiting incoming traffic from untrusted hosts.

On Linux systems, the `iptables` interface provides access to the underlying `netfilter` firewall. On Windows systems, `netsh` command line interface provides access to the underlying Windows Firewall. For additional information about firewall configuration, see *Configure Linux iptables Firewall for MongoDB* (page 331) and *Configure Windows netsh Firewall for MongoDB* (page 334).

For best results and to minimize overall exposure, ensure that *only* traffic from trusted sources can reach `mongod` and `mongos` instances and that the `mongod` and `mongos` instances can only connect to trusted outputs.

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2[13] 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 TLS/SSL rather than IPSEC VPNs for performance issues.

Depending on configuration and implementation, VPNs provide for certificate validation and a choice of encryption protocols, which requires a rigorous level of authentication and identification of all clients. Furthermore, because VPNs provide a secure tunnel, by using a VPN connection to control access to your MongoDB instance, you can prevent tampering and "man-in-the-middle" attacks.

6.2.5 Security and MongoDB API Interfaces

On this page

The following section contains strategies to limit risks related to MongoDB's available interfaces including JavaScript, HTTP, and REST interfaces.

[13] https://docs.mongodb.org/ecosystem/platforms/amazon-ec2

JavaScript and the Security of the mongo Shell

The following JavaScript evaluation behaviors of the mongo shell represents risk exposures.

JavaScript Expression or JavaScript File

The mongo program can evaluate JavaScript expressions using the command line *--eval* option. Also, the mongo program can evaluate a JavaScript file (.js) passed directly to it (e.g. mongo someFile.js).

Because the mongo program evaluates the JavaScript directly, inputs should only come from trusted sources.

.mongorc.js File

If a .mongorc.js file exists [14], the mongo shell will evaluate a .mongorc.js file before starting. You can disable this behavior by passing the *mongo --norc* option.

HTTP Status Interface

The HTTP status interface provides a web-based interface that includes a variety of operational data, logs, and status reports regarding the mongod or mongos instance. The HTTP interface is always available on the port numbered 1000 greater than the primary mongod port. By default, the HTTP interface port is 28017, but is indirectly set using the port option which allows you to configure the primary mongod port.

Without the net.http.RESTInterfaceEnabled 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 enabled run time option or the *--nohttpinterface* command line option. See also *Configuration Options* (page 323).

REST API

The REST API to MongoDB provides additional information and write access on top of the HTTP Status interface. While the REST API does not provide any support for insert, update, or remove operations, it does provide administrative access, and its accessibility represents a vulnerability in a secure environment. The REST interface is *disabled* by default, and is not recommended for production use.

If you must use the REST API, please control and limit access to the REST API. The REST API does not include any support for authentication, even when running with authorization enabled.

See the following documents for instructions on restricting access to the REST API interface:

- *Configure Linux iptables Firewall for MongoDB* (page 331)
- *Configure Windows netsh Firewall for MongoDB* (page 334)

6.2.6 Auditing

On this page

[14] On Linux and Unix systems, mongo reads the .mongorc.js file from *$HOME*/.mongorc.js (i.e. ~/.mongorc.js). On Windows, mongo.exe reads the .mongorc.js file from *%HOME%*.mongorc.js or *%HOMEDRIVE%%HOMEPATH%*.mongorc.js.

New in version 2.6.

MongoDB Enterprise includes an auditing capability for `mongod` and `mongos` instances. The auditing facility allows administrators and users to track system activity for deployments with multiple users and applications. The auditing facility can write audit events to the console, the *syslog*, a JSON file, or a BSON file.

Audit Events and Filter

To enable auditing for MongoDB Enterprise, see *Configure System Events Auditing* (page 397).

Once enabled, the auditing system can record the following operations:

- schema (DDL),
- replica set,
- authentication and authorization, and
- general operations.

For details on the audit log messages, see *System Event Audit Messages* (page 424).

By default, the auditing system records all these operations; however, you can *set up filters* (page 399) to restrict the events captured. To set up filters, see *Configure Audit Filters* (page 399).

Audit Guarantee

The auditing system writes every audit event [15] to an in-memory buffer of audit events. MongoDB writes this buffer to disk periodically. For events collected from any single connection, the events have a total order: if MongoDB writes one event to disk, the system guarantees that it has written all prior events for that connection to disk.

If an audit event entry corresponds to an operation that affects the durable state of the database, such as a modification to data, MongoDB will always write the audit event to disk *before* writing to the *journal* for that entry.

That is, before adding an operation to the journal, MongoDB writes all audit events on the connection that triggered the operation, up to and including the entry for the operation.

These auditing guarantees require that MongoDB run with `journaling` enabled.

Warning: MongoDB may lose events **if** the server terminates before it commits the events to the audit log. The client may receive confirmation of the event before MongoDB commits to the audit log. For example, while auditing an aggregation operation, the server might crash after returning the result but before the audit log flushes.

6.2.7 Kerberos Authentication

On this page

New in version 2.4.

[15] Audit configuration can include a *filter* (page 399) to limit events to audit.

Overview

MongoDB Enterprise provides support for Kerberos authentication of MongoDB clients to `mongod` and `mongos`. Kerberos is an industry standard authentication protocol for large client/server systems. Kerberos allows MongoDB and applications to take advantage of existing authentication infrastructure and processes.

Kerberos Components and MongoDB

Principals

In a Kerberos-based system, every participant in the authenticated communication is known as a "principal", and every principal must have a unique name.

Principals belong to administrative units called *realms*. For each realm, the Kerberos Key Distribution Center (KDC) maintains a database of the realm's principal and the principals' associated "secret keys".

For a client-server authentication, the client requests from the KDC a "ticket" for access to a specific asset. KDC uses the client's secret and the server's secret to construct the ticket which allows the client and server to mutually authenticate each other, while keeping the secrets hidden.

For the configuration of MongoDB for Kerberos support, two kinds of principal names are of interest: *user principals* (page 327) and *service principals* (page 327).

User Principal To authenticate using Kerberos, you must add the Kerberos user principals to MongoDB to the `$external` database. User principal names have the form:

```
<username>@<KERBEROS REALM>
```

For every user you want to authenticate using Kerberos, you must create a corresponding user in MongoDB in the `$external` database.

For examples of adding a user to MongoDB as well as authenticating as that user, see *Configure MongoDB with Kerberos Authentication on Linux* (page 369) and *Configure MongoDB with Kerberos Authentication on Windows* (page 372).

See also:

User and Role Management Tutorials (page 381) for general information regarding creating and managing users in MongoDB.

Service Principal Every MongoDB `mongod` and `mongos` instance (or `mongod.exe` or `mongos.exe` on Windows) must have an associated service principal. Service principal names have the form:

```
<service>/<fully qualified domain name>@<KERBEROS REALM>
```

For MongoDB, the `<service>` defaults to `mongodb`. For example, if `m1.example.com` is a MongoDB server, and `example.com` maintains the `EXAMPLE.COM` Kerberos realm, then `m1` should have the service principal name `mongodb/m1.example.com@EXAMPLE.COM`.

To specify a different value for `<service>`, use `serviceName` during the start up of `mongod` or `mongos` (or `mongod.exe` or `mongos.exe`). `mongo` shell or other clients may also specify a different service principal name using `serviceName`.

Service principal names must be reachable over the network using the fully qualified domain name (FQDN) part of its service principal name.

By default, Kerberos attempts to identify hosts using the `/etc/krb5.conf` file before using DNS to resolve hosts.

On Windows, if running MongoDB as a service, see *Assign Service Principal Name to MongoDB Windows Service* (page 374).

Linux Keytab Files

Linux systems can store Kerberos authentication keys for a *service principal* (page 327) in *keytab* files. Each Kerberized `mongod` and `mongos` instance running on Linux must have access to a keytab file containing keys for its *service principal* (page 327).

To keep keytab files secure, use file permissions that restrict access to only the user that runs the `mongod` or `mongos` process.

Tickets

On Linux, MongoDB clients can use Kerberos's `kinit` program to initialize a credential cache for authenticating the user principal to servers.

Windows Active Directory

Unlike on Linux systems, `mongod` and `mongos` instances running on Windows do not require access to keytab files. Instead, the `mongod` and `mongos` instances read their server credentials from a credential store specific to the operating system.

However, from the Windows Active Directory, you can export a keytab file for use on Linux systems. See Ktpass[16] for more information.

Authenticate With Kerberos

To configure MongoDB for Kerberos support and authenticate, see *Configure MongoDB with Kerberos Authentication on Linux* (page 369) and *Configure MongoDB with Kerberos Authentication on Windows* (page 372).

Operational Considerations

The HTTP Console

The MongoDB HTTP Console[17] interface does not support Kerberos authentication.

DNS

Each host that runs a `mongod` or `mongos` instance must have both `A` and `PTR` DNS records to provide forward and reverse lookup.

Without `A` and `PTR` DNS records, the host cannot resolve the components of the Kerberos domain or the Key Distribution Center (KDC).

[16] http://technet.microsoft.com/en-us/library/cc753771.aspx
[17] https://docs.mongodb.org/ecosystem/tools/http-interfaces/#http-console

System Time Synchronization

To successfully authenticate, the system time for each `mongod` and `mongos` instance must be within 5 minutes of the system time of the other hosts in the Kerberos infrastructure.

Kerberized MongoDB Environments

Driver Support

The following MongoDB drivers support Kerberos authentication:

- Java[18]
- C#[19]
- C++[20]
- Python[21]

Use with Additional MongoDB Authentication Mechanism

Although MongoDB supports the use of Kerberos authentication with other authentication mechanisms, only add the other mechanisms as necessary. See the `Incorporate Additional Authentication Mechanisms` section in *Configure MongoDB with Kerberos Authentication on Linux* (page 369) and *Configure MongoDB with Kerberos Authentication on Windows* (page 372) for details.

Additional Resources

- MongoDB LDAP and Kerberos Authentication with Dell (Quest) Authentication Services[22]
- MongoDB with Red Hat Enterprise Linux Identity Management and Kerberos[23]

6.3 Security Tutorials

The following tutorials provide instructions for enabling and using the security features available in MongoDB.

Network Security Tutorials **(page 330)** Ensure that the underlying network configuration supports a secure operating environment for MongoDB deployments, and appropriately limits access to MongoDB deployments.

Configure Linux iptables Firewall for MongoDB **(page 331)** Basic firewall configuration patterns and examples for `iptables` on Linux systems.

Configure Windows netsh Firewall for MongoDB **(page 334)** Basic firewall configuration patterns and examples for `netsh` on Windows systems.

Configure mongod and mongos for TLS/SSL **(page 338)** TLS/SSL allows MongoDB clients to support encrypted connections to `mongod` instances.

[18] https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-java-driver/
[19] https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-csharp-driver/
[20] https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-cpp-driver/
[21] http://api.mongodb.org/python/current/examples/authentication.html
[22] https://www.mongodb.com/blog/post/mongodb-ldap-and-kerberos-authentication-dell-quest-authentication-services?jmp=docs
[23] http://docs.mongodb.org/ecosystem/tutorial/manage-red-hat-enterprise-linux-identity-management/?jmp=docs

Continue reading from *Network Security Tutorials* (page 330) for more information on running MongoDB in secure environments.

Security Deployment Tutorials **(page 348)** These tutorials describe procedures for deploying MongoDB using authentication and authorization.

Access Control Tutorials **(page 352)** These tutorials describe procedures relevant for the configuration, operation, and maintenance of MongoDB's access control system.

Enable Client Access Control **(page 353)** Describes the process for enabling authentication for MongoDB deployments.

Use x.509 Certificates to Authenticate Clients **(page 357)** Use x.509 for client authentication.

Use x.509 Certificate for Membership Authentication **(page 359)** Use x.509 for internal member authentication for replica sets and sharded clusters.

Configure MongoDB with Kerberos Authentication on Linux **(page 369)** For MongoDB Enterprise Linux, describes the process to enable Kerberos-based authentication for MongoDB deployments.

Continue reading from *Access Control Tutorials* (page 352) for additional tutorials on configuring MongoDB's authentication systems.

Enable Authentication after Creating the User Administrator **(page 355)** Describes an alternative process for enabling authentication for MongoDB deployments.

User and Role Management Tutorials **(page 381)** MongoDB's access control system provides a flexible role-based access control system that you can use to limit access to MongoDB deployments. The tutorials in this section describe the configuration an setup of the authorization system.

Add a User to a Database **(page 383)** Create non-administrator users using MongoDB's role-based authentication system.

Create a Role **(page 386)** Create custom role.

Modify a User's Access **(page 391)** Modify the actions available to a user on specific database resources.

View Roles **(page 393)** View a role's privileges.

Continue reading from *User and Role Management Tutorials* (page 381) for additional tutorials on managing users and privileges in MongoDB's authorization system.

Auditing Tutorials **(page 397)** MongoDB Enterprise provides auditing of operations. The tutorials in this section describe procedures to enable and configure the auditing feature.

Create a Vulnerability Report **(page 402)** Report a vulnerability in MongoDB.

6.3.1 Network Security Tutorials

The following tutorials provide information on handling network security for MongoDB.

Configure Linux iptables Firewall for MongoDB **(page 331)** Basic firewall configuration patterns and examples for `iptables` on Linux systems.

Configure Windows netsh Firewall for MongoDB **(page 334)** Basic firewall configuration patterns and examples for `netsh` on Windows systems.

Configure mongod and mongos for TLS/SSL **(page 338)** TLS/SSL allows MongoDB clients to support encrypted connections to `mongod` instances.

TLS/SSL Configuration for Clients **(page 342)** Configure clients to connect to MongoDB instances that use TLS/SSL.

Upgrade a Cluster to Use TLS/SSL **(page 346)** Rolling upgrade process to use TLS/SSL.

Configure MongoDB for FIPS **(page 347)** Configure for Federal Information Processing Standard (FIPS).

Configure Linux `iptables` Firewall for MongoDB

On this page

On contemporary Linux systems, the `iptables` program provides methods for managing the Linux Kernel's `netfilter` or network packet filtering capabilities. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic firewall configurations for `iptables` firewalls on Linux. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see *Security Concepts* (page 316).

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2[24] 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 323) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod` and `mongos` instances.

Be aware that, by default, the default policy of `iptables` is to allow all connections and traffic unless explicitly disabled. The configuration changes outlined in this document will create rules that explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed. When you have properly configured your `iptables` rules to allow only the traffic that you want to permit, you can *Change Default Policy to DROP* (page 333).

Patterns

This section contains a number of patterns and examples for configuring `iptables` for use with MongoDB deployments. If you have configured different ports using the `port` configuration setting, you will need to modify the rules accordingly.

[24] https://docs.mongodb.org/ecosystem/platforms/amazon-ec2

Traffic to and from `mongod` Instances This pattern is applicable to all `mongod` instances running as standalone instances or as part of a *replica set*.

The goal of this pattern is to explicitly allow traffic to the `mongod` instance from the application server. In the following examples, replace `<ip-address>` with the IP address of the application server:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27017 -m state --state NEW,ESTABLISHED -
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27017 -m state --state ESTABLISHED -j ACCEPT
```

The first rule allows all incoming traffic from `<ip-address>` on port `27017`, which allows the application server to connect to the `mongod` instance. The second rule, allows outgoing traffic from the `mongod` to reach the application server.

Optional

If you have only one application server, you can replace `<ip-address>` with either the IP address itself, such as: `198.51.100.55`. You can also express this using CIDR notation as `198.51.100.55/32`. If you want to permit a larger block of possible IP addresses you can allow traffic from a `/24` using one of the following specifications for the `<ip-address>`, as follows:

```
10.10.10.10/24
10.10.10.10/255.255.255.0
```

Traffic to and from `mongos` Instances `mongos` instances provide query routing for *sharded clusters*. Clients connect to `mongos` instances, which behave from the client's perspective as `mongod` instances. In turn, the `mongos` connects to all `mongod` instances that are components of the sharded cluster.

Use the same `iptables` command to allow traffic to and from these instances as you would from the `mongod` instances that are members of the replica set. Take the configuration outlined in the *Traffic to and from mongod Instances* (page 332) 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. [25] Config servers listen for connections on port `27019`. As a result, add the following `iptables` rules to the config server to allow incoming and outgoing connection on port `27019`, for connection to the other config servers.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT
```

Replace `<ip-address>` with the address or address space of *all* the `mongod` that provide config servers.

Additionally, config servers need to allow incoming connections from all of the `mongos` instances in the cluster *and* all `mongod` instances in the cluster. Add rules that resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -
```

Replace `<ip-address>` with the address of the `mongos` instances and the shard `mongod` instances.

Traffic to and from a MongoDB Shard Server For shard servers, running as *`mongod --shardsvr`* [26] Because the default port number is `27018` when running with the `shardsvr` value for the `clusterRole` setting, you must configure the following `iptables` rules to allow traffic to and from each shard:

[25] You also can run a config server by using the `configsvr` value for the `clusterRole` setting in a configuration file.

[26] You can also specify the shard server option with the `shardsvr` value for the `clusterRole` setting in the configuration file. Shard members are also often conventional replica sets using the default port.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27018 -m state --state NEW,ESTABLISHED -
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

Replace the `<ip-address>` specification with the IP address of all `mongod`. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members, to:

- all `mongod` instances in the shard's replica sets.
- all `mongod` instances in other shards. [27]

Furthermore, shards need to be able make outgoing connections to:

- all `mongod` instances in the config servers.

Create a rule that resembles the following, and replace the `<ip-address>` with the address of the config servers and the `mongos` instances:

```
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT
```

Provide Access For Monitoring Systems

1. The `mongostat` diagnostic tool, when running with the `--discover` needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the `mongos` instances.
2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

   ```
   iptables -A INPUT -s <ip-address> -p tcp --destination-port 28017 -m state --state NEW,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 config server `mongod` instances running with the `shardsvr` value for the `clusterRole` setting, the rule would resemble the following:

   ```
   iptables -A INPUT -s <ip-address> -p tcp --destination-port 28018 -m state --state NEW,ESTABLISH
   ```

   For config server `mongod` instances running with the `configsvr` value for the `clusterRole` setting, 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
```

[27] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

Manage and Maintain `iptables` Configuration

This section contains a number of basic operations for managing and using `iptables`. There are various front end tools that automate some aspects of `iptables` configuration, but at the core all `iptables` front ends provide the same basic functionality:

Make all `iptables` Rules Persistent By default all `iptables` rules are only stored in memory. When your system restarts, your firewall rules will revert to their defaults. When you have tested a rule set and have guaranteed that it effectively controls traffic you can use the following operations to you should make the rule set persistent.

On Red Hat Enterprise Linux, Fedora Linux, and related distributions you can issue the following command:

```
service iptables save
```

On Debian, Ubuntu, and related distributions, you can use the following command to dump the `iptables` rules to the `/etc/iptables.conf` file:

```
iptables-save > /etc/iptables.conf
```

Run the following operation to restore the network rules:

```
iptables-restore < /etc/iptables.conf
```

Place this command in your `rc.local` file, or in the `/etc/network/if-up.d/iptables` file with other similar operations.

List all `iptables` Rules To list all of currently applied `iptables` rules, use the following operation at the system shell.

```
iptables -L
```

Flush all `iptables` Rules If you make a configuration mistake when entering `iptables` rules or simply need to revert to the default rule set, you can use the following operation at the system shell to flush all rules:

```
iptables -F
```

If you've already made your `iptables` rules persistent, you will need to repeat the appropriate procedure in the *Make all iptables Rules Persistent* (page 334) section.

Configure Windows `netsh` Firewall for MongoDB

On this page

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 Concepts* (page 316).

See also:

Windows Firewall[28] 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 323) of all MongoDB processes, you must configure networking rules that permit *only* required communication between your application and the appropriate `mongod.exe` and `mongos.exe` instances.

The configuration changes outlined in this document will create rules which explicitly allow traffic from specific addresses and on specific ports, using a default policy that drops all traffic that is not explicitly allowed.

You can configure the *Windows Firewall* with using the `netsh` command line tool or through a windows application. On Windows Server 2008 this application is *Windows Firewall With Advanced Security* in *Administrative Tools*. On previous versions of Windows Server, access the *Windows Firewall* application in the *System and Security* control panel.

The procedures in this document use the `netsh` command line tool.

Patterns

This section contains a number of patterns and examples for configuring *Windows Firewall* for use with MongoDB deployments. If you have configured different ports using the `port` configuration setting, you will need to modify the rules accordingly.

Traffic to and from `mongod.exe` Instances This pattern is applicable to all `mongod.exe` instances running as standalone instances or as part of a *replica set*. The goal of this pattern is to explicitly allow traffic to the `mongod.exe` instance from the application server.

```
netsh advfirewall firewall add rule name="Open mongod port 27017" dir=in action=allow protocol=TCP lo
```

This rule allows all incoming traffic to port `27017`, which allows the application server to connect to the `mongod.exe` instance.

Windows Firewall also allows enabling network access for an entire application rather than to a specific port, as in the following example:

```
netsh advfirewall firewall add rule name="Allowing mongod" dir=in action=allow program=" C:\mongodb\b
```

You can allow all access for a `mongos.exe` server, with the following invocation:

[28] http://technet.microsoft.com/en-us/network/bb545423.aspx

```
netsh advfirewall firewall add rule name="Allowing mongos" dir=in action=allow program=" C:\mongodb\
```

Traffic to and from `mongos.exe` Instances `mongos.exe` instances provide query routing for *sharded clusters*. Clients connect to `mongos.exe` instances, which behave from the client's perspective as `mongod.exe` instances. In turn, the `mongos.exe` connects to all `mongod.exe` instances that are components of the sharded cluster.

Use the same *Windows Firewall* command to allow traffic to and from these instances as you would from the `mongod.exe` instances that are members of the replica set.

```
netsh advfirewall firewall add rule name="Open mongod shard port 27018" dir=in action=allow protocol=
```

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. [29] Configuration servers listen for connections on port `27019`. As a result, add the following *Windows Firewall* rules to the config server to allow incoming and outgoing connection on port `27019`, for connection to the other config servers.

```
netsh advfirewall firewall add rule name="Open mongod config svr port 27019" dir=in action=allow prot
```

Additionally, config servers need to allow incoming connections from all of the `mongos.exe` instances in the cluster *and* all `mongod.exe` instances in the cluster. Add rules that resemble the following:

```
netsh advfirewall firewall add rule name="Open mongod config svr inbound" dir=in action=allow protoco
```

Replace `<ip-address>` with the addresses of the `mongos.exe` instances and the shard `mongod.exe` instances.

Traffic to and from a MongoDB Shard Server For shard servers, running as *`mongod --shardsvr`* [30] Because the default port number is `27018` when running with the `shardsvr` value for the `clusterRole` setting, you must configure the following *Windows Firewall* rules to allow traffic to and from each shard:

```
netsh advfirewall firewall add rule name="Open mongod shardsvr inbound" dir=in action=allow protocol=
netsh advfirewall firewall add rule name="Open mongod shardsvr outbound" dir=out action=allow protoco
```

Replace the `<ip-address>` specification with the IP address of all `mongod.exe` instances. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members to:

- all `mongod.exe` instances in the shard's replica sets.
- all `mongod.exe` instances in other shards. [31]

Furthermore, shards need to be able make outgoing connections to:

- all `mongos.exe` instances.
- all `mongod.exe` instances in the config servers.

Create a rule that resembles the following, and replace the `<ip-address>` with the address of the config servers and the `mongos.exe` instances:

```
netsh advfirewall firewall add rule name="Open mongod config svr outbound" dir=out action=allow proto
```

[29] You also can run a config server by using the `configsrv` value for the `clusterRole` setting in a configuration file.

[30] You can also specify the shard server option with the `shardsvr` value for the `clusterRole` setting in the configuration file. Shard members are also often conventional replica sets using the default port.

[31] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

Provide Access For Monitoring Systems

1. The `mongostat` diagnostic tool, when running with the `--discover` needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the `mongos.exe` instances.

2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

   ```
   netsh advfirewall firewall add rule name="Open mongod HTTP monitoring inbound" dir=in action=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 config server `mongod` instances running with the `shardsvr` value for the `clusterRole` setting, 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` instances running with the `configsvr` value for the `clusterRole` setting, the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongod configsvr HTTP monitoring inbound" dir=in
   ```

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` 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"
```

Configure mongod and mongos for TLS/SSL

On this page

Overview

This document helps you to configure MongoDB to support TLS/SSL. MongoDB clients can use TLS/SSL to encrypt connections to `mongod` and `mongos` instances. MongoDB TLS/SSL implementation uses OpenSSL libraries.

Note: Although TLS is the successor to SSL, this page uses the more familiar term SSL to refer to TLS/SSL.

These instructions assume that you have already installed a build of MongoDB that includes SSL support and that your client driver supports SSL. For instructions on upgrading a cluster currently not using SSL to using SSL, see *Upgrade a Cluster to Use TLS/SSL* (page 346).

Changed in version 2.6: MongoDB's SSL encryption only allows use of strong SSL ciphers with a minimum of 128-bit key length for all connections.

New in version 2.6: MongoDB Enterprise for Windows includes support for SSL.

Prerequisites

Important: A full description of TLS/SSL, PKI (Public Key Infrastructure) certificates, and Certificate Authority is beyond the scope of this document. This page assumes prior knowledge of TLS/SSL as well as access to valid certificates.

MongoDB Support The default distribution of MongoDB[32] 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[33].

Client Support See *TLS/SSL Configuration for Clients* (page 342) to learn about SSL support for Python, Java, Ruby, and other clients.

Certificate Authorities For production use, your MongoDB deployment should use valid certificates generated and signed by a single certificate authority. You or your organization can generate and maintain an independent certificate authority, or use certificates generated by a third-party SSL vendor. Obtaining and managing certificates is beyond the scope of this documentation.

[32] http://www.mongodb.org/downloads
[33] http://www.mongodb.com/products/mongodb-enterprise

`.pem` File Before you can use SSL, you must have a `.pem` file containing a public key certificate and its associated private key.

MongoDB can use any valid SSL certificate issued by a certificate authority, or a self-signed certificate. If you use a self-signed certificate, although the communications channel will be encrypted, there will be *no* validation of server identity. Although such a situation will prevent eavesdropping on the connection, it leaves you vulnerable to a man-in-the-middle attack. Using a certificate signed by a trusted certificate authority will permit MongoDB drivers to verify the server's identity.

In general, avoid using self-signed certificates unless the network is trusted.

Additionally, with regards to *authentication among replica set/sharded cluster members* (page 318), in order to minimize exposure of the private key and allow hostname validation, it is advisable to use different certificates on different servers.

For *testing* purposes, you can generate a self-signed certificate and private key on a Unix system with a command that resembles the following:

```
cd /etc/ssl/
openssl req -newkey rsa:2048 -new -x509 -days 365 -nodes -out mongodb-cert.crt -keyout mongodb-cert.k
```

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
```

See also:

Use x.509 Certificates to Authenticate Clients (page 357)

Procedures

Set Up `mongod` and `mongos` with SSL Certificate and Key To use SSL in your MongoDB deployment, include the following run-time options with `mongod` and `mongos`:

- `net.ssl.mode` set to `requireSSL`. This setting restricts each server to use only SSL encrypted connections. You can also specify either the value `allowSSL` or `preferSSL` to set up the use of mixed SSL modes on a port. See `net.ssl.mode` for details.
- `PEMKeyfile` with the `.pem` file that contains the SSL certificate and key.

Consider the following syntax for `mongod`:

```
mongod --sslMode requireSSL --sslPEMKeyFile <pem>
```

For example, given an SSL certificate located at `/etc/ssl/mongodb.pem`, configure `mongod` to use SSL encryption for all connections with the following command:

```
mongod --sslMode requireSSL --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 passphrase. See *SSL Certificate Passphrase* (page 341).

You may also specify these options in the `configuration file`, as in the following examples:

If using the `YAML configuration file format`:

```
net:
   ssl:
      mode: requireSSL
      PEMKeyFile: /etc/ssl/mongodb.pem
```

Or, if using the older older configuration file format[34]:

```
sslMode = requireSSL
sslPEMKeyFile = /etc/ssl/mongodb.pem
```

To connect, to `mongod` and `mongos` instances using SSL, the `mongo` shell and MongoDB tools must include the `--ssl` option. See *TLS/SSL Configuration for Clients* (page 342) for more information on connecting to `mongod` and `mongos` running with SSL.

See also:

Upgrade a Cluster to Use TLS/SSL (page 346)

Set Up `mongod` and `mongos` with Certificate Validation To set up `mongod` or `mongos` for SSL encryption using an SSL certificate signed by a certificate authority, include the following run-time options during startup:

- `net.ssl.mode` set to `requireSSL`. This setting restricts each server to use only SSL encrypted connections. You can also specify either the value `allowSSL` or `preferSSL` to set up the use of mixed SSL modes on a port. See `net.ssl.mode` for details.
- `PEMKeyfile` with the name of the `.pem` file that contains the signed SSL certificate and key.
- `CAFile` with the name of the `.pem` file that contains the root certificate chain from the Certificate Authority.

Consider the following syntax for `mongod`:

```
mongod --sslMode requireSSL --sslPEMKeyFile <pem> --sslCAFile <ca>
```

For example, given a signed SSL certificate located at `/etc/ssl/mongodb.pem` and the certificate authority file at `/etc/ssl/ca.pem`, you can configure `mongod` for SSL encryption as follows:

```
mongod --sslMode requireSSL --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 passphrase. See *SSL Certificate Passphrase* (page 341).

You may also specify these options in the `configuration file`, as in the following examples:

If using the `YAML configuration file format`:

```
net:
   ssl:
      mode: requireSSL
      PEMKeyFile: /etc/ssl/mongodb.pem
      CAFile: /etc/ssl/ca.pem
```

Or, if using the older older configuration file format[35]:

[34] http://docs.mongodb.org/v2.4/reference/configuration-options
[35] http://docs.mongodb.org/v2.4/reference/configuration-options

```
sslMode = requireSSL
sslPEMKeyFile = /etc/ssl/mongodb.pem
sslCAFile = /etc/ssl/ca.pem
```

To connect, to `mongod` and `mongos` instances using SSL, the `mongo` tools must include the both the `--ssl` and `--sslPEMKeyFile` option. See *TLS/SSL Configuration for Clients* (page 342) for more information on connecting to `mongod` and `mongos` running with SSL.

See also:

Upgrade a Cluster to Use TLS/SSL (page 346)

Block Revoked Certificates for Clients To prevent clients with revoked certificates from connecting, include the `sslCRLFile` to specify a `.pem` file that contains revoked certificates.

For example, the following `mongod` with SSL configuration includes the `sslCRLFile` setting:

```
mongod --sslMode requireSSL --sslCRLFile /etc/ssl/ca-crl.pem --sslPEMKeyFile /etc/ssl/mongodb.pem --s
```

Clients with revoked certificates in the `/etc/ssl/ca-crl.pem` will not be able to connect to this `mongod` instance.

Validate Only if a Client Presents a Certificate In most cases it is important to ensure that clients present valid certificates. However, if you have clients that cannot present a client certificate, or are transitioning to using a certificate authority you may only want to validate certificates from clients that present a certificate.

If you want to bypass validation for clients that don't present certificates, include the `weakCertificateValidation` run-time option with `mongod` and `mongos`. If the client does not present a certificate, no validation occurs. These connections, though not validated, are still encrypted using SSL.

For example, consider the following `mongod` with an SSL configuration that includes the `weakCertificateValidation` setting:

```
mongod --sslMode requireSSL --sslWeakCertificateValidation --sslPEMKeyFile /etc/ssl/mongodb.pem --ssl
```

Then, clients can connect either with the option `--ssl` and **no** certificate or with the option `--ssl` and a **valid** certificate. See *TLS/SSL Configuration for Clients* (page 342) 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.

SSL Certificate Passphrase The PEM files for `PEMKeyfile` and `ClusterFile` may be encrypted. With encrypted PEM files, you must specify the passphrase at startup with a command-line or a configuration file option or enter the passphrase when prompted.

Changed in version 2.6: In previous versions, you can only specify the passphrase with a command-line or a configuration file option.

To specify the passphrase in clear text on the command line or in a configuration file, use the `PEMKeyPassword` and/or the `ClusterPassword` option.

To have MongoDB prompt for the passphrase at the start of `mongod` or `mongos` and avoid specifying the passphrase in clear text, omit the `PEMKeyPassword` and/or the `ClusterPassword` option. MongoDB will prompt for each passphrase as necessary.

Important: The passphrase prompt option is available if you run the MongoDB instance in the foreground with

a connected terminal. If you run `mongod` or `mongos` in a non-interactive session (e.g. without a terminal or as a service on Windows), you cannot use the passphrase prompt option.

Run in FIPS Mode See *Configure MongoDB for FIPS* (page 347) for more details.

TLS/SSL Configuration for Clients

On this page

Clients must have support for TLS/SSL to work with a `mongod` or a `mongos` instance that has TLS/SSL support enabled. The current versions of the Python, Java, Ruby, Node.js, .NET, and C++ drivers have support for TLS/SSL, with full support coming in future releases of other drivers.

Important: A full description of TLS/SSL, PKI (Public Key Infrastructure) certificates, and Certificate Authority is beyond the scope of this document. This page assumes prior knowledge of TLS/SSL as well as access to valid certificates.

Note: Although TLS is the successor to SSL, this page uses the more familiar term SSL to refer to TLS/SSL.

See also:

Configure mongod and mongos for TLS/SSL (page 338).

`mongo` Shell SSL Configuration

For SSL connections, you must use the `mongo` shell built with SSL support or distributed with MongoDB Enterprise. To support SSL, `mongo` has the following settings:

- `--ssl`
- `--sslPEMKeyFile` with the name of the `.pem` file that contains the SSL certificate and key.
- `--sslCAFile` with the name of the `.pem` file that contains the certificate from the Certificate Authority (CA).

  Warning: If the `mongo` shell or any other tool that connects to `mongos` or `mongod` is run without `--sslCAFile`, it will not attempt to validate server certificates. This results in vulnerability to expired `mongod` and `mongos` certificates as well as to foreign processes posing as valid `mongod` or `mongos` instances. Ensure that you *always* specify the CA file against which server certificates should be validated in cases where intrusion is a possibility.

- `--sslPEMKeyPassword` option if the client certificate-key file is encrypted.

Connect to MongoDB Instance with SSL Encryption To connect to a `mongod` or `mongos` instance that requires *only a SSL encryption mode* (page 339), start `mongo` shell with `--ssl`, as in the following:

```
mongo --ssl
```

Connect to MongoDB Instance that Requires Client Certificates To connect to a `mongod` or `mongos` that requires *CA-signed client certificates* (page 340), start the `mongo` shell with `--ssl` and the `--sslPEMKeyFile` option to specify the signed certificate-key file, as in the following:

```
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Connect to MongoDB Instance that Validates when Presented with a Certificate To connect to a `mongod` or `mongos` instance that *only requires valid certificates when the client presents a certificate* (page 341), start `mongo` shell either with the `--ssl` ssl and **no** certificate or with the `--ssl` ssl and a **valid** signed certificate.

For example, if `mongod` is running with weak certificate validation, both of the following `mongo` shell clients can connect to that `mongod`:

```
mongo --ssl
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem
```

Important: If the client presents a certificate, the certificate must be valid.

MongoDB Cloud Manager

The MongoDB Cloud Manager Monitoring agent will also have to connect via SSL in order to gather its statistics. Because the agent already utilizes SSL for its communications to the MongoDB Cloud Manager servers, this is just a matter of enabling SSL support in MongoDB Cloud Manager itself on a per host basis.

See the MongoDB Cloud Manager documentation[36] for more information about SSL configuration.

PyMongo

Add the "`ssl=True`" parameter to a PyMongo `MongoClient`[37] 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
```

For more details, see the Python MongoDB Driver page[38].

[36] https://docs.cloud.mongodb.com/
[37] http://api.mongodb.org/python/current/api/pymongo/mongo_client.html#pymongo.mongo_client.MongoClient
[38] https://docs.mongodb.org/ecosystem/drivers/python

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() );
    }
}
```

For more details, see the Java MongoDB Driver page[39].

Ruby

The recent versions of the Ruby driver have support for connections to SSL servers. Install the latest version of the driver with the following command:

```
gem install mongo
```

Then connect to a standalone instance, using the following form:

```
require 'rubygems'
require 'mongo'

connection = MongoClient.new('localhost', 27017, :ssl => true)
```

Replace `connection` with the following if you're connecting to a replica set:

```
connection = MongoReplicaSetClient.new(['localhost:27017'],
                                       ['localhost:27018'],
                                       :ssl => true)
```

Here, `mongod` instance run on "`localhost:27017`" and "`localhost:27018`".

For more details, see the Ruby MongoDB Driver page[40].

Node.JS (`node-mongodb-native`)

In the node-mongodb-native[41] driver, use the following invocation to connect to a `mongod` or `mongos` instance via SSL:

[39] https://docs.mongodb.org/ecosystem/drivers/java
[40] https://docs.mongodb.org/ecosystem/drivers/ruby
[41] https://github.com/mongodb/node-mongodb-native

```
var db1 = new Db(MONGODB, new Server("127.0.0.1", 27017,
                                      { auto_reconnect: false, poolSize:4, ssl:true } );
```

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:true}
);
```

For more details, see the Node.JS MongoDB Driver page[42].

.NET

As of release 1.6, the .NET driver supports SSL connections with `mongod` and `mongos` instances. To connect using SSL, you must add an option to the connection string, specifying `ssl=true` as follows:

```
var connectionString = "mongodb://localhost/?ssl=true";
var server = MongoServer.Create(connectionString);
```

The .NET driver will validate the certificate against the local trusted certificate store, in addition to providing encryption of the server. This behavior may produce issues during testing if the server uses a self-signed certificate. If you encounter this issue, add the `sslverifycertificate=false` option to the connection string to prevent the .NET driver from validating the certificate, as follows:

```
var connectionString = "mongodb://localhost/?ssl=true&sslverifycertificate=false";
var server = MongoServer.Create(connectionString);
```

For more details, see the .NET MongoDB Driver page[43].

MongoDB Tools

Changed in version 2.6.

Various MongoDB utility programs supports SSL. These tools include:

- `mongodump`
- `mongoexport`
- `mongofiles`
- `mongoimport`
- `mongooplog`
- `mongorestore`
- `mongostat`
- `mongotop`

To use SSL connections with these tools, use the same SSL options as the `mongo` shell. See *mongo Shell SSL Configuration* (page 342).

[42] https://docs.mongodb.org/ecosystem/drivers/node-js
[43] https://docs.mongodb.org/ecosystem/drivers/csharp

Upgrade a Cluster to Use TLS/SSL

The default distribution of MongoDB[44] does **not** contain support for TLS/SSL. To use TLS/SSL you can either compile MongoDB with TLS/SSL support or use MongoDB Enterprise. See *Configure mongod and mongos for TLS/SSL* (page 338) for more information about TLS/SSL and MongoDB.

Important: A full description of TLS/SSL, PKI (Public Key Infrastructure) certificates, and Certificate Authority is beyond the scope of this document. This page assumes prior knowledge of TLS/SSL as well as access to valid certificates.

Changed in version 2.6.

The MongoDB server supports listening for both TLS/SSL encrypted and unencrypted connections on the same TCP port. This allows upgrades of MongoDB clusters to use TLS/SSL encrypted connections.

To upgrade from a MongoDB cluster using no TLS/SSL encryption to one using *only* TLS/SSL encryption, use the following rolling upgrade process:

1. For each node of a cluster, start the node with the option `--sslMode` set to `allowSSL`. The `--sslMode allowSSL` setting allows the node to accept both TLS/SSL and non-TLS/non-SSL incoming connections. Its connections to other servers do not use TLS/SSL. Include other *TLS/SSL options* (page 338) as well as any other options that are required for your specific configuration. For example:

   ```
   mongod --replSet <name> --sslMode allowSSL --sslPEMKeyFile <path to TLS/SSL Certificate and key
   ```

   Upgrade all nodes of the cluster to these settings.

   You may also specify these options in the `configuration file`. If using a `YAML format configuration file`, specify the following settings in the file:

   ```
   net:
      ssl:
         mode: <disabled|allowSSL|preferSSL|requireSSL>
         PEMKeyFile: <path to TLS/SSL certificate and key PEM file>
         CAFile: <path to root CA PEM file>
   ```

   Or, if using the older configuration file format[45]:

   ```
   sslMode = <disabled|allowSSL|preferSSL|requireSSL>
   sslPEMKeyFile = <path to TLS/SSL certificate and key PEM file>
   sslCAFile = <path to root CA PEM file>
   ```

2. Switch all clients to use TLS/SSL. See *TLS/SSL Configuration for Clients* (page 342).

3. For each node of a cluster, use the `setParameter` command to update the `sslMode` to `preferSSL`. [46] With `preferSSL` as its `net.ssl.mode`, the node accepts both TLS/SSL and non-TLS/non-SSL incoming connections, and its connections to other servers use TLS/SSL. For example:

   ```
   db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "preferSSL" } )
   ```

   Upgrade all nodes of the cluster to these settings.

   At this point, all connections should be using TLS/SSL.

4. For each node of the cluster, use the `setParameter` command to update the `sslMode` to `requireSSL`. [1] With `requireSSL` as its `net.ssl.mode`, the node will reject any non-TLS/non-SSL connections. For example:

[44] http://www.mongodb.org/downloads
[45] http://docs.mongodb.org/v2.4/reference/configuration-options
[46] As an alternative to using the `setParameter` command, you can also restart the nodes with the appropriate TLS/SSL options and values.

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "requireSSL" } )
```

5. After the upgrade of all nodes, edit the `configuration file` with the appropriate TLS/SSL settings to ensure that upon subsequent restarts, the cluster uses TLS/SSL.

Configure MongoDB for FIPS

On this page

New in version 2.6.

Overview

The Federal Information Processing Standard (FIPS) is a U.S. government computer security standard used to certify software modules and libraries that encrypt and decrypt data securely. You can configure MongoDB to run with a FIPS 140-2 certified library for OpenSSL. Configure FIPS to run by default or as needed from the command line.

Prerequisites

Important: A full description of FIPS and TLS/SSL is beyond the scope of this document. This tutorial assumes prior knowledge of FIPS and TLS/SSL.

Only the MongoDB Enterprise[47] version supports FIPS mode. See *Install MongoDB Enterprise* (page 27) to download and install MongoDB Enterprise[48] to use FIPS mode.

Your system must have an OpenSSL library configured with the FIPS 140-2 module. At the command line, type `openssl version` to confirm your OpenSSL software includes FIPS support.

For Red Hat Enterprise Linux 6.x (RHEL 6.x) or its derivatives such as CentOS 6.x, the OpenSSL toolkit must be at least `openssl-1.0.1e-16.el6_5` to use FIPS mode. To upgrade the toolkit for these platforms, issue the following command:

```
sudo yum update openssl
```

Some versions of Linux periodically execute a process to *prelink* dynamic libraries with pre-assigned addresses. This process modifies the OpenSSL libraries, specifically `libcrypto`. The OpenSSL FIPS mode will subsequently fail the signature check performed upon startup to ensure `libcrypto` has not been modified since compilation.

To configure the Linux prelink process to not prelink `libcrypto`:

```
sudo bash -c "echo '-b /usr/lib64/libcrypto.so.*' >>/etc/prelink.conf.d/openssl-prelink.conf"
```

[47] http://www.mongodb.com/products/mongodb-enterprise
[48] http://www.mongodb.com/products/mongodb-enterprise

Considerations

FIPS is property of the encryption system and not the access control system. However, if your environment requires FIPS compliant encryption *and* access control, you must ensure that the access control system uses only FIPS-compliant encryption.

MongoDB's FIPS support covers the way that MongoDB uses OpenSSL for network encryption and X509 authentication. If you use Kerberos or LDAP Proxy authentication, you muse ensure that these external mechanisms are FIPS-compliant. `MONGODB-CR` authentication is *not* FIPS compliant.

Procedure

Configure MongoDB to use TLS/SSL See *Configure mongod and mongos for TLS/SSL* (page 338) for details about configuring OpenSSL.

Run `mongod` or `mongos` instance in FIPS mode Perform these steps after you *Configure mongod and mongos for TLS/SSL* (page 338).

Step 1: Change configuration file. To configure your `mongod` or `mongos` instance to use FIPS mode, shut down the instance and update the configuration file with the following setting:

```
net:
   ssl:
      FIPSMode: true
```

Step 2: Start `mongod` or `mongos` instance with configuration file. For example, run this command to start the `mongod` instance with its configuration file:

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

Confirm FIPS mode is running Check the server log file for a message FIPS is active:

```
FIPS 140-2 mode activated
```

6.3.2 Security Deployment Tutorials

The following tutorials provide information in deploying MongoDB using authentication and authorization.

Deploy Replica Set and Configure Authentication and Authorization **(page 348)** Configure a replica set that has authentication enabled.

Deploy Replica Set and Configure Authentication and Authorization

On this page

Overview

With *authentication* (page 316) enabled, MongoDB forces all clients to identify themselves before granting access to the server. *Authorization* (page 320), in turn, allows administrators to define and limit the resources and operations that a user can access. Using authentication and authorization is a key part of a complete security strategy.

All MongoDB deployments support authentication. By default, MongoDB does not require authorization checking. You can enforce authorization checking when deploying MongoDB, or on an existing deployment; however, you cannot enable authorization checking on a running deployment without downtime.

This tutorial provides a procedure for creating a MongoDB *replica set* (page 563) that uses the challenge-response authentication mechanism. The tutorial includes creation of a minimal authorization system to support basic operations.

Considerations

Authentication In this procedure, you will configure MongoDB using the default challenge-response authentication mechanism, using the `keyFile` to supply the password for *inter-process authentication* (page 318). The content of the key file is the shared secret used for all internal authentication.

All deployments that enforce authorization checking should have one *user administrator* user that can create new users and modify existing users. During this procedure you will create a user administrator that you will use to administer this deployment.

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See *Replica Set Deployment Architectures* (page 575) for more information.

Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongod.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

Procedure

This procedure deploys a replica set in which all members use the same key file.

Step 1: Start one member of the replica set. This `mongod` should *not* enable `auth`.

Step 2: Create administrative users. The following operations will create two users: a user administrator that will be able to create and modify users (`siteUserAdmin`), and a `root` (page 412) user (`siteRootAdmin`) that you will use to complete the remainder of the tutorial:

```
use admin
db.createUser( {
    user: "siteUserAdmin",
    pwd: "<password>",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  });
db.createUser( {
    user: "siteRootAdmin",
    pwd: "<password>",
    roles: [ { role: "root", db: "admin" } ]
  });
```

Step 3: Stop the `mongod` instance.

Step 4: Create the key file to be used by each member of the replica set. Create the key file your deployment will use to authenticate servers to each other.

To generate pseudo-random data to use for a `keyfile`, issue the following `openssl` command:

```
openssl rand -base64 741 > mongodb-keyfile
chmod 600 mongodb-keyfile
```

You may generate a key file using any method you choose. Always ensure that the password stored in the key file is both long and contains a high amount of entropy. Using `openssl` in this manner helps generate such a key.

Step 5: Copy the key file to each member of the replica set. Copy the `mongodb-keyfile` to all hosts where components of a MongoDB deployment run. Set the permissions of these files to `600` so that only the *owner* of the file can read or write this file to prevent other users on the system from accessing the shared secret.

Step 6: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the key file and the name of the replica set. Also specify other parameters as needed for your deployment. For replication-specific parameters, see *cli-mongod-replica-set* required by your deployment.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies parameters through the *`--keyFile`* and *`--replSet`* command-line options:

```
mongod --keyFile /mysecretdirectory/mongodb-keyfile --replSet "rs0"
```

The following example specifies parameters through a configuration file:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 7: Connect to the member of the replica set where you created the administrative users. Connect to the replica set member you started and authenticate as the `siteRootAdmin` user. From the `mongo` shell, use the following operation to authenticate:

```
use admin
db.auth("siteRootAdmin", "<password>");
```

Step 8: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 9: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 659):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 10: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

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

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 11: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Step 12: Create additional users to address operational requirements. You can use *built-in roles* (page 405) to create common types of database users, such as the `dbOwner` (page 407) role to create a database administrator, the `readWrite` (page 405) role to create a user who can update data, or the `read` (page 405) role to create user who can search data but no more. You also can define *custom roles* (page 321).

For example, the following creates a database administrator for the `products` database:

```
use products
db.createUser(
  {
    user: "productsDBAdmin",
    pwd: "password",
    roles:
    [
      {
        role: "dbOwner",
        db: "products"
      }
    ]
  }
)
```

For an overview of roles and privileges, see *Authorization* (page 320). For more information on adding users, see *Add a User to a Database* (page 383).

6.3.3 Access Control Tutorials

The following tutorials provide instructions for MongoDB"s authentication and authorization related features.

Enable Client Access Control **(page 353)** Describes the process for enabling authentication for MongoDB deployments.

Enable Authentication in a Sharded Cluster **(page 354)** Control access to a sharded cluster through a key file and the `keyFile` setting on each of the cluster's components.

Enable Authentication after Creating the User Administrator **(page 355)** Describes an alternative process for enabling authentication for MongoDB deployments.

Use x.509 Certificates to Authenticate Clients **(page 357)** Use x.509 for client authentication.

Use x.509 Certificate for Membership Authentication **(page 359)** Use x.509 for internal member authentication for replica sets and sharded clusters.

Authenticate Using SASL and LDAP with ActiveDirectory **(page 363)** Describes the process for authentication using SASL/LDAP with ActiveDirectory.

Authenticate Using SASL and LDAP with OpenLDAP **(page 366)** Describes the process for authentication using SASL/LDAP with OpenLDAP.

Configure MongoDB with Kerberos Authentication on Linux **(page 369)** For MongoDB Enterprise Linux, describes the process to enable Kerberos-based authentication for MongoDB deployments.

Configure MongoDB with Kerberos Authentication on Windows **(page 372)** For MongoDB Enterprise for Windows, describes the process to enable Kerberos-based authentication for MongoDB deployments.

Authenticate to a MongoDB Instance or Cluster **(page 375)** Describes the process for authenticating to MongoDB systems using the `mongo` shell.

Generate a Key File **(page 376)** Use key file to allow the components of MongoDB sharded cluster or replica set to mutually authenticate.

Troubleshoot Kerberos Authentication on Linux **(page 377)** Steps to troubleshoot Kerberos-based authentication for MongoDB deployments.

Implement Field Level Redaction **(page 379)** Describes the process to set up and access document content that can have different access levels for the same data.

Enable Client Access Control

On this page

Overview

Enabling access control on a MongoDB instance restricts access to the instance by requiring that users identify themselves when connecting. In this procedure, you enable access control and then create the instance's first user, which must be a user administrator. The user administrator grants further access to the instance by creating additional users.

Considerations

If you create the user administrator before enabling access control, MongoDB disables the *localhost exception* (page 319). In that case, you must use the "*Enable Authentication after Creating the User Administrator* (page 355)" procedure to enable access control.

This procedure uses the *localhost exception* (page 319) to allow you to create the first user after enabling authentication. See *Localhost Exception* (page 319) and *Authentication* (page 316) for more information.

Procedure

Step 1: Start the MongoDB instance with authentication enabled. Start the `mongod` or `mongos` instance with the `authorization` or `keyFile` setting. Use `authorization` on a standalone instance. Use `keyFile` on an instance in a *replica set* or *sharded cluster*.

For example, to start a `mongod` with authentication enabled and a key file stored in `/private/var`, first set the following option in the `mongod`'s configuration file:

```
security:
  keyFile: /private/var/key.pem
```

Then start the `mongod` and specify the config file. For example:

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

After you enable authentication, only the user administrator can connect to the MongoDB instance. The user administrator must log in and grant further access to the instance by creating additional users.

Step 2: Connect to the MongoDB instance via the localhost exception. Connect to the MongoDB instance from a client running on the same system. This access is made possible by the *localhost exception* (page 319).

Step 3: Create the system user administrator. Add the user with the `userAdminAnyDatabase` (page 411) role, and only that role.

The following example creates the user `siteUserAdmin` user on the `admin` database:

```
use admin
db.createUser(
  {
    user: "siteUserAdmin",
    pwd: "password",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

After you create the user administrator, the *localhost exception* (page 319) is no longer available.

The `mongo` shell executes a number of commands at start up. As a result, when you log in as the user administrator, you may see authentication errors from one or more commands. You may ignore these errors, which are expected, because the `userAdminAnyDatabase` (page 411) role does not have permissions to run some of the start up commands.

Step 4: Create additional users. Login in with the user administrator's credentials and create additional users. See *Add a User to a Database* (page 383).

Next Steps

If you need to disable access control for any reason, restart the process without the `authorization` or `keyFile` setting.

Enable Authentication in a Sharded Cluster

On this page

New in version 2.0: Support for authentication with sharded clusters.

Overview

When authentication is enabled on a sharded cluster every client that accesses the cluster must provide credentials. This includes MongoDB instances that access each other within the cluster.

To enable authentication on a sharded cluster, you must enable authentication individually on each component of the cluster. This means enabling authentication on each `mongos` and each `mongod`, including each config server, and all members of a shard's replica set.

Authentication requires an authentication mechanism and, in most cases, a `keyfile`. The content of the key file must be the same on all cluster members.

Consideration

It is not possible to convert an existing sharded cluster that does not enforce access control to require authentication without taking all components of the cluster offline for a short period of time.

Procedure

Step 1: Create a key file. Create the key file your deployment will use to authenticate servers to each other.

To generate pseudo-random data to use for a `keyfile`, issue the following `openssl` command:

```
openssl rand -base64 741 > mongodb-keyfile
chmod 600 mongodb-keyfile
```

You may generate a key file using any method you choose. Always ensure that the password stored in the key file is both long and contains a high amount of entropy. Using `openssl` in this manner helps generate such a key.

Step 2: Enable authentication on each component in the cluster. On each `mongos` and `mongod` in the cluster, including all config servers and shards, specify the key file using one of the following approaches:

Specify the key file in the configuration file. In the configuration file, set the `keyFile` option to the key file's path and then start the component, as in the following example:

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

Specify the key file at runtime. When starting the component, set the *`--keyFile`* option, which is an option for both `mongos` instances and `mongod` instances. Set the *`--keyFile`* to the key file's path. The `keyFile` setting implies the `authorization` setting, which means in most cases you do not need to set `authorization` explicitly.

Step 3: Add users. While connected to a `mongos`, add the first administrative user and then add subsequent users. See *Create a User Administrator* (page 381).

Related Documents

- *Authentication* (page 316)
- *Security* (page 313)
- *Use x.509 Certificate for Membership Authentication* (page 359)

Enable Authentication after Creating the User Administrator

On this page

Overview

Enabling authentication on a MongoDB instance restricts access to the instance by requiring that users identify themselves when connecting. In this procedure, you will create the instance's first user, which must be a user administrator and then enable authentication. Then, you can authenticate as the user administrator to create additional users and grant additional access to the instance.

This procedures outlines how enable authentication after creating the user administrator. The approach requires a restart. To enable authentication without restarting, see *Enable Client Access Control* (page 353).

Considerations

This document outlines a procedure for enabling authentication for MongoDB instance where you create the first user on an existing MongoDB system that does not require authentication before restarting the instance and requiring authentication. You can use the *localhost exception* (page 319) to gain access to a system with no users and authentication enabled. See *Enable Client Access Control* (page 353) for the description of that procedure.

Procedure

Step 1: Start the MongoDB instance without authentication. Start the `mongod` or `mongos` instance *without* the `authorization` or `keyFile` setting. For example:

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

For details on starting a `mongod` or `mongos`, see *Manage mongod Processes* (page 236) or *Deploy a Sharded Cluster* (page 705).

Step 2: Create the system user administrator. Add the user with the `userAdminAnyDatabase` (page 411) role, and only that role.

The following example creates the user `siteUserAdmin` user on the `admin` database:

```
use admin
db.createUser(
  {
    user: "siteUserAdmin",
    pwd: "password",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

Step 3: Re-start the MongoDB instance with authentication enabled. Re-start the `mongod` or `mongos` instance with the `authorization` or `keyFile` setting. Use `authorization` on a standalone instance. Use `keyFile` on an instance in a *replica set* or *sharded cluster*.

The following example enables authentication on a standalone `mongod` using the `authorization` command-line option:

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

Step 4: Create additional users. Log in with the user administrator's credentials and create additional users. See *Add a User to a Database* (page 383).

Next Steps

If you need to disable authentication for any reason, restart the process without the `authorization` or `keyFile` option.

Use x.509 Certificates to Authenticate Clients

On this page

New in version 2.6.

MongoDB supports x.509 certificate authentication for use with a secure *TLS/SSL connection* (page 338). The x.509 client authentication allows *clients to authenticate to servers with certificates* (page 357) rather than with a username and password.

To use x.509 authentication for the internal authentication of replica set/sharded cluster members, see *Use x.509 Certificate for Membership Authentication* (page 359).

Prerequisites

Important: A full description of TLS/SSL, PKI (Public Key Infrastructure) certificates, in particular x.509 certificates, and Certificate Authority is beyond the scope of this document. This tutorial assumes prior knowledge of TLS/SSL as well as access to valid x.509 certificates.

Certificate Authority For production use, your MongoDB deployment should use valid certificates generated and signed by a single certificate authority. You or your organization can generate and maintain an independent certificate authority, or use certificates generated by a third-party SSL vendor. Obtaining and managing certificates is beyond the scope of this documentation.

Client x.509 Certificate 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
```

- Each unique MongoDB user must have a unique certificate.
- A client x.509 certificate's subject, which contains the Distinguished Name (`DN`), must **differ** from that of a *Member x.509 Certificate* (page 360). Specifically, the subjects must differ with regards to at least one of the following attributes: Organization (`O`), the Organizational Unit (`OU`) or the Domain Component (`DC`).

  Warning: If a client x.509 certificate's subject has the same `O`, `OU`, and `DC` combination as the *Member x.509 Certificate* (page 360), the client will be identified as a cluster member and granted full permission on the system.

Procedures

Configure MongoDB Server

Use Command-line Options You can configure the MongoDB server from the command line, e.g.:

```
mongod --clusterAuthMode x509 --sslMode requireSSL --sslPEMKeyFile <path to SSL certificate and key P
```

> **Warning:** If the *--sslCAFile* option and its target file are not specified, x.509 client and member authentication will not function. `mongod`, and `mongos` in sharded systems, will not be able to verify the certificates of processes connecting to it against the trusted certificate authority (CA) that issued them, breaking the certificate chain.
> As of version 2.6.4, `mongod` will not start with x.509 authentication enabled if the CA file is not specified.

Use Configuration File You may also specify these options in the `configuration file`.

Starting in MongoDB 2.6, you can specify the configuration for MongoDB in `YAML format`, e.g.:

```
security:
   clusterAuthMode: x509
net:
   ssl:
      mode: requireSSL
      PEMKeyFile: <path to TLS/SSL certificate and key PEM file>
      CAFile: <path to root CA PEM file>
```

For backwards compatibility, you can also specify the configuration using the older configuration file format[49], e.g.:

```
clusterAuthMode = x509
sslMode = requireSSL
sslPEMKeyFile = <path to TLS/SSL certificate and key PEM file>
sslCAFile = <path to the root CA PEM file>
```

Include any additional options, TLS/SSL or otherwise, that are required for your specific configuration.

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. Each unique x.509 client certificate corresponds to a single MongoDB user; i.e. you cannot use a single client certificate to authenticate more than one 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` shell, to add the user with both the `readWrite` role in the `test` database and the `userAdminAnyDatabase` role which is defined only in the `admin` database:

[49] http://docs.mongodb.org/v2.4/reference/configuration-options

```
db.getSiblingDB("$external").runCommand(
  {
    createUser: "CN=myName,OU=myOrgUnit,O=myOrg,L=myLocality,ST=myState,C=myCountry",
    roles: [
             { role: 'readWrite', db: 'test' },
             { role: 'userAdminAnyDatabase', db: 'admin' }
           ],
    writeConcern: { w: "majority" , wtimeout: 5000 }
  }
)
```

In the above example, to add the user with the `readWrite` role in the `test` database, the role specification document specified `'test'` in the `db` field. To add `userAdminAnyDatabase` role for the user, the above example specified `'admin'` in the `db` field.

Note: Some roles are defined only in the `admin` database, including: `clusterAdmin`, `readAnyDatabase`, `readWriteAnyDatabase`, `dbAdminAnyDatabase`, and `userAdminAnyDatabase`. To add a user with these roles, specify `'admin'` in the `db`.

See *Add a User to a Database* (page 383) for details on adding a user with roles.

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 358).

To authenticate, use the `db.auth()` method in the `$external` database, specifying `"MONGODB-X509"` for the `mechanism` field, and the *user that corresponds to the client certificate* (page 358) for the `user` field.

For example, if using the `mongo` shell,

1. Connect `mongo` shell to the `mongod` set up for TLS/SSL:

   ```
   mongo --ssl --sslPEMKeyFile <path to CA signed client PEM file> --sslCAFile <path to root CA PEM
   ```

2. To perform the authentication, use the `db.auth()` 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.

   ```
   db.getSiblingDB("$external").auth(
     {
       mechanism: "MONGODB-X509",
       user: "CN=myName,OU=myOrgUnit,O=myOrg,L=myLocality,ST=myState,C=myCountry"
     }
   )
   ```

Use x.509 Certificate for Membership Authentication

On this page

New in version 2.6.

MongoDB supports x.509 certificate authentication for use with a secure *TLS/SSL connection* (page 338). Sharded cluster members and replica set members can use x.509 certificates to verify their membership to the cluster or the replica set instead of using *keyfiles* (page 316). The membership authentication is an internal process.

For client authentication with x.509, see *Use x.509 Certificates to Authenticate Clients* (page 357).

Important: A full description of TLS/SSL, PKI (Public Key Infrastructure) certificates, in particular x.509 certificates, and Certificate Authority is beyond the scope of this document. This tutorial assumes prior knowledge of TLS/SSL as well as access to valid x.509 certificates.

Member x.509 Certificate

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 Distinguished Name (`DN`), found in the member certificate's `subject`, must specify a non-empty value for *at least one* of the following attributes: Organization (`O`), the Organizational Unit (`OU`) or the Domain Component (`DC`).
- The Organization attributes (`O`'s), the Organizational Unit attributes (`OU`'s), and the Domain Components (`DC`'s) must match those from the certificates for the other cluster members. To match, the certificate must match all specifications of these attributes, or even the non-specification of these attributes. The order of the attributes does not matter.

  In the following example, the two `DN`'s contain matching specifications for `O`, `OU` as well as the non-specification of the `DC` attribute.

  ```
  CN=host1,OU=Dept1,O=MongoDB,ST=NY,C=US
  C=US, ST=CA, O=MongoDB, OU=Dept1, CN=host2
  ```

  However, the following two `DN`'s contain a mismatch for the `OU` attribute since one contains two `OU` specifications and the other, only one specification.

  ```
  CN=host1,OU=Dept1,OU=Sales,O=MongoDB
  CN=host2,OU=Dept1,O=MongoDB
  ```

- Either the Common Name (`CN`) or one of the Subject Alternative Name (`SAN`) entries must match the hostname of the server, used by the other members of the cluster.

  For example, the certificates for a cluster could have the following subjects:

  ```
  subject= CN=<myhostname1>,OU=Dept1,O=MongoDB,ST=NY,C=US
  subject= CN=<myhostname2>,OU=Dept1,O=MongoDB,ST=NY,C=US
  subject= CN=<myhostname3>,OU=Dept1,O=MongoDB,ST=NY,C=US
  ```

You *can* use an x509 certificate that does not have Extended Key Usage (EKU) attributes set. If you use EKU attribute in the `PEMKeyFile` certificate, then specify the `clientAuth` and/or `serverAuth` attributes (i.e. "TLS Web Client Authentication" and "TLS Web Server Authentication,") as needed. The certificate that you specify for the `PEMKeyFile` option requires the `serverAuth` attribute, and the certificate you specify to `clusterFile` requires the `clientAuth` attribute. If you omit `ClusterFile`, `mongod` will use the certificate specified to `PEMKeyFile` for member authentication.

Configure Replica Set/Sharded Cluster

Use Command-line Options To specify the x.509 certificate for internal cluster member authentication, append the additional TLS/SSL options `--clusterAuthMode` and `--sslClusterFile`, as in the following example for a member of a replica set:

```
mongod --replSet <name> --sslMode requireSSL --clusterAuthMode x509 --sslClusterFile <path to members
```

Include any additional options, TLS/SSL or otherwise, that are required for your specific configuration. For instance, if the membership key is encrypted, set the `--sslClusterPassword` to the passphrase to decrypt the key or have MongoDB prompt for the passphrase. See *SSL Certificate Passphrase* (page 341) for details.

Warning: If the `--sslCAFile` option and its target file are not specified, x.509 client and member authentication will not function. `mongod`, and `mongos` in sharded systems, will not be able to verify the certificates of processes connecting to it against the trusted certificate authority (CA) that issued them, breaking the certificate chain.
As of version 2.6.4, `mongod` will not start with x.509 authentication enabled if the CA file is not specified.

Use Configuration File You can specify the configuration for MongoDB in a `YAML formatted` configuration file, as in the following example:

```
security:
   clusterAuthMode: x509
net:
   ssl:
      mode: requireSSL
      PEMKeyFile: <path to TLS/SSL certificate and key PEM file>
      CAFile: <path to root CA PEM file>
      clusterFile: <path to x.509 membership certificate and key PEM file>
```

See `security.clusterAuthMode`, `net.ssl.mode`, `net.ssl.PEMKeyFile`, `net.ssl.CAFile`, and `net.ssl.clusterFile` for more information on the settings.

Upgrade from Keyfile Authentication to x.509 Authentication

To upgrade clusters that are currently using keyfile authentication to x.509 authentication, use a rolling upgrade process.

Clusters Currently Using TLS/SSL For clusters using TLS/SSL and keyfile authentication, to upgrade to x.509 cluster authentication, use the following rolling upgrade process:

1. For each node of a cluster, start the node with the option `--clusterAuthMode` set to `sendKeyFile` and the option `--sslClusterFile` set to the appropriate path of the node's certificate. Include other *TLS/SSL options* (page 338) as well as any other options that are required for your specific configuration. For example:

   ```
   mongod --replSet <name> --sslMode requireSSL --clusterAuthMode sendKeyFile --sslClusterFile <pat
   ```

   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 an 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, connect to the node and use the `setParameter` command to update the `clusterAuthMode` to `sendX509`. [50] For example,

[50] As an alternative to using the `setParameter` command, you can also restart the nodes with the appropriate TLS/SSL and x509 options and values.

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, clusterAuthMode: "sendX509" } )
```

With this setting, each node uses its x.509 certificate, specified with the *--sslClusterFile* option in the previous step, to authenticate itself as a member. However, each node continues to accept either a keyfile or an 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, connect to the node and use the `setParameter` command to update the `clusterAuthMode` to `x509` to only use the x.509 certificate for authentication. [1] For example:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, clusterAuthMode: "x509" } )
```

4. After the upgrade of all nodes, edit the `configuration file` with the appropriate x.509 settings to ensure that upon subsequent restarts, the cluster uses x.509 authentication.

See *--clusterAuthMode* for the various modes and their descriptions.

Clusters Currently Not Using TLS/SSL For clusters using keyfile authentication but not TLS/SSL, to upgrade to x.509 authentication, use the following rolling upgrade process:

1. For each node of a cluster, start the node with the option *--sslMode* set to `allowSSL`, the option *--clusterAuthMode* set to `sendKeyFile` and the option *--sslClusterFile* set to the appropriate path of the node's certificate. Include other *TLS/SSL options* (page 338) as well as any other options that are required for your specific configuration. For example:

```
mongod --replSet <name> --sslMode allowSSL --clusterAuthMode sendKeyFile --sslClusterFile <path
```

The *--sslMode allowSSL* setting allows the node to accept both TLS/SSL and non-TLS/non-SSL incoming connections. Its outgoing connections do not use TLS/SSL.

The *--clusterAuthMode sendKeyFile* setting allows each node continues to use its keyfile to authenticate itself as a member. However, each node can now accept either a keyfile or an x.509 certificate from other members to authenticate those members.

Upgrade all nodes of the cluster to these settings.

2. Then, for each node of a cluster, connect to the node and use the `setParameter` command to update the `sslMode` to `preferSSL` and the `clusterAuthMode` to `sendX509`. [1] For example:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "preferSSL", clusterAuthMode: "
```

With the `sslMode` set to `preferSSL`, the node accepts both TLS/SSL and non-TLS/non-SSL incoming connections, and its outgoing connections use TLS/SSL.

With the `clusterAuthMode` set to `sendX509`, each node uses its x.509 certificate, specified with the *--sslClusterFile* option in the previous step, to authenticate itself as a member. However, each node continues to accept either a keyfile or an x.509 certificate from other members to authenticate those members.

Upgrade all nodes of the cluster to these settings.

3. Optional but recommended. Finally, for each node of the cluster, connect to the node and use the `setParameter` command to update the `sslMode` to `requireSSL` and the `clusterAuthMode` to `x509`. [1] For example:

```
db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "requireSSL", clusterAuthMode:
```

With the `sslMode` set to `requireSSL`, the node only uses TLS/SSLs connections.

With the `clusterAuthMode` set to `x509`, the node only uses the x.509 certificate for authentication.

4. After the upgrade of all nodes, edit the `configuration file` with the appropriate TLS/SSL and x.509 settings to ensure that upon subsequent restarts, the cluster uses x.509 authentication.

See *`--clusterAuthMode`* for the various modes and their descriptions.

Authenticate Using SASL and LDAP with ActiveDirectory

On this page

MongoDB Enterprise provides support for proxy authentication of users. This allows administrators to configure a MongoDB cluster to authenticate users by proxying authentication requests to a specified Lightweight Directory Access Protocol (LDAP) service.

Considerations

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB does **not** support LDAP authentication in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 847) for upgrade instructions.

Use secure encrypted or trusted connections between clients and the server, as well as between `saslauthd` and the LDAP server. The LDAP server uses the `SASL PLAIN` mechanism, sending and receiving data in **plain text**. You should use only a trusted channel such as a VPN, a connection encrypted with TLS/SSL, or a trusted wired network.

Configure `saslauthd`

LDAP support for user authentication requires proper configuration of the `saslauthd` daemon process as well as the MongoDB server.

Step 1: Specify the mechanism. On systems that configure `saslauthd` with the `/etc/sysconfig/saslauthd` file, such as Red Hat Enterprise Linux, Fedora, CentOS, and Amazon Linux AMI, set the mechanism `MECH` to `ldap`:

```
MECH=ldap
```

On systems that configure `saslauthd` with the `/etc/default/saslauthd` file, such as Ubuntu, set the `MECHANISMS` option to `ldap`:

```
MECHANISMS="ldap"
```

Step 2: Adjust caching behavior. On certain Linux distributions, `saslauthd` starts with the caching of authentication credentials *enabled*. Until restarted or until the cache expires, `saslauthd` will not contact the LDAP server to re-authenticate users in its authentication cache. This allows `saslauthd` to successfully authenticate users in its cache, even in the LDAP server is down or if the cached users' credentials are revoked.

To set the expiration time (in seconds) for the authentication cache, see the -t option[51] of `saslauthd`.

[51] http://www.linuxcommand.org/man_pages/saslauthd8.html

Step 3: Configure LDAP Options with ActiveDirectory. If the `saslauthd.conf` file does not exist, create it. The `saslauthd.conf` file usually resides in the `/etc` folder. If specifying a different file path, see the -O option[52] of `saslauthd`.

To use with ActiveDirectory, start `saslauthd` with the following configuration options set in the `saslauthd.conf` file:

```
ldap_servers: <ldap uri>
ldap_use_sasl: yes
ldap_mech: DIGEST-MD5
ldap_auth_method: fastbind
```

For the `<ldap uri>`, specify the uri of the ldap server. For example, `ldap_servers: ldaps://ad.example.net`.

For more information on `saslauthd` configuration, see http://www.openldap.org/doc/admin24/guide.html#Configuringsaslauthd.

Step 4: Test the `saslauthd` configuration. Use `testsaslauthd` utility to test the `saslauthd` configuration. For example:

```
testsaslauthd -u testuser -p testpassword -f /var/run/saslauthd/mux
```

Configure MongoDB

Step 1: Add user to MongoDB for authentication. Add the user to the `$external` database in MongoDB. To specify the user's privileges, assign *roles* (page 320) to the user.

For example, the following adds a user with read-only access to the `records` database.

```
db.getSiblingDB("$external").createUser(
    {
      user : <username>,
      roles: [ { role: "read", db: "records" } ]
    }
)
```

Add additional principals as needed. For more information about creating and managing users, see `http://docs.mongodb.org/manual/reference/command/nav-user-management`.

Step 2: Configure MongoDB server. To configure the MongoDB server to use the `saslauthd` instance for proxy authentication, start the `mongod` with the following options:

- *`--auth`*,
- `authenticationMechanisms` parameter set to `PLAIN`, and
- `saslauthdPath` parameter set to the path to the Unix-domain Socket of the `saslauthd` instance.

Configure the MongoDB server using either the command line option `--setParameter` or the `configuration file`. Specify additional configurations as appropriate for your configuration.

If you use the `authorization` option to enforce authentication, you will need privileges to create a user.

[52] http://www.linuxcommand.org/man_pages/saslauthd8.html

Use specific `saslauthd` socket path. For socket path of `/<some>/<path>/saslauthd`, set the `saslauthdPath` to `/<some>/<path>/saslauthd/mux`, as in the following command line example:

```
mongod --auth --setParameter saslauthdPath=/<some>/<path>/saslauthd/mux --setParameter authentication
```

Or if using a `YAML format configuration file`, specify the following settings in the file:

```
security:
   authorization: enabled

setParameter:
   saslauthdPath: /<some>/<path>/saslauthd/mux
   authenticationMechanisms: PLAIN
```

Or, if using the older configuration file format[53]:

```
auth=true
setParameter=saslauthdPath=/<some>/<path>/saslauthd/mux
setParameter=authenticationMechanisms=PLAIN
```

Use default Unix-domain socket path. To use the default Unix-domain socket path, set the `saslauthdPath` to the empty string `""`, as in the following command line example:

```
mongod --auth --setParameter saslauthdPath="" --setParameter authenticationMechanisms=PLAIN
```

Or if using a `YAML format configuration file`, specify the following settings in the file:

```
security:
   authorization: enabled

setParameter:
   saslauthdPath: ""
   authenticationMechanisms: PLAIN
```

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

```
auth=true
setParameter=saslauthdPath=""
setParameter=authenticationMechanisms=PLAIN
```

Step 3: Authenticate the user in the `mongo` shell. To perform the authentication in the `mongo` shell, use the `db.auth()` method in the `$external` database.

Specify the value `"PLAIN"` in the `mechanism` field, the user and password in the `user` and `pwd` fields respectively, and the value `false` in the `digestPassword` field. You **must** specify `false` for `digestPassword` since the server must receive an undigested password to forward on to `saslauthd`, as in the following example:

```
db.getSiblingDB("$external").auth(
   {
     mechanism: "PLAIN",
     user: <username>,
     pwd:  <cleartext password>,
     digestPassword: false
   }
)
```

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

The server forwards the password in plain text. In general, use only on a trusted channel (VPN, TLS/SSL, trusted wired network). See Considerations.

Authenticate Using SASL and LDAP with OpenLDAP

On this page

MongoDB Enterprise provides support for proxy authentication of users. This allows administrators to configure a MongoDB cluster to authenticate users by proxying authentication requests to a specified Lightweight Directory Access Protocol (LDAP) service.

Considerations

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB does **not** support LDAP authentication in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 847) for upgrade instructions.

Use secure encrypted or trusted connections between clients and the server, as well as between `saslauthd` and the LDAP server. The LDAP server uses the `SASL PLAIN` mechanism, sending and receiving data in **plain text**. You should use only a trusted channel such as a VPN, a connection encrypted with TLS/SSL, or a trusted wired network.

Configure `saslauthd`

LDAP support for user authentication requires proper configuration of the `saslauthd` daemon process as well as the MongoDB server.

Step 1: Specify the mechanism. On systems that configure `saslauthd` with the `/etc/sysconfig/saslauthd` file, such as Red Hat Enterprise Linux, Fedora, CentOS, and Amazon Linux AMI, set the mechanism `MECH` to `ldap`:

```
MECH=ldap
```

On systems that configure `saslauthd` with the `/etc/default/saslauthd` file, such as Ubuntu, set the `MECHANISMS` option to `ldap`:

```
MECHANISMS="ldap"
```

Step 2: Adjust caching behavior. On certain Linux distributions, `saslauthd` starts with the caching of authentication credentials *enabled*. Until restarted or until the cache expires, `saslauthd` will not contact the LDAP server to re-authenticate users in its authentication cache. This allows `saslauthd` to successfully authenticate users in its cache, even in the LDAP server is down or if the cached users' credentials are revoked.

To set the expiration time (in seconds) for the authentication cache, see the -t option[55] of `saslauthd`.

[55] http://www.linuxcommand.org/man_pages/saslauthd8.html

Step 3: Configure LDAP Options with OpenLDAP. If the `saslauthd.conf` file does not exist, create it. The `saslauthd.conf` file usually resides in the `/etc` folder. If specifying a different file path, see the -O option[56] of `saslauthd`.

To connect to an OpenLDAP server, update the `saslauthd.conf` file with the following configuration options:

```
ldap_servers: <ldap uri>
ldap_search_base: <search base>
ldap_filter: <filter>
```

The `ldap_servers` specifies the uri of the LDAP server used for authentication. In general, for OpenLDAP installed on the local machine, you can specify the value `ldap://localhost:389` or if using LDAP over TLS/SSL, you can specify the value `ldaps://localhost:636`.

The `ldap_search_base` specifies distinguished name to which the search is relative. The search includes the base or objects below.

The `ldap_filter` specifies the search filter.

The values for these configuration options should correspond to the values specific for your test. For example, to filter on email, specify `ldap_filter: (mail=%n)` instead.

OpenLDAP Example A sample `saslauthd.conf` file for OpenLDAP includes the following content:

```
ldap_servers: 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`) under the domain components (`dc`) `example` and `com`.

For more information on `saslauthd` configuration, see http://www.openldap.org/doc/admin24/guide.html#Configuringsaslauthd.

Step 4: Test the `saslauthd` configuration. Use `testsaslauthd` utility to test the `saslauthd` configuration. For example:

```
testsaslauthd -u testuser -p testpassword -f /var/run/saslauthd/mux
```

Configure MongoDB

Step 1: Add user to MongoDB for authentication. Add the user to the `$external` database in MongoDB. To specify the user's privileges, assign *roles* (page 320) to the user.

For example, the following adds a user with read-only access to the `records` database.

```
db.getSiblingDB("$external").createUser(
    {
      user : <username>,
      roles: [ { role: "read", db: "records" } ]
    }
)
```

Add additional principals as needed. For more information about creating and managing users, see `http://docs.mongodb.org/manual/reference/command/nav-user-management`.

[56] http://www.linuxcommand.org/man_pages/saslauthd8.html

Step 2: Configure MongoDB server. To configure the MongoDB server to use the `saslauthd` instance for proxy authentication, start the `mongod` with the following options:

- `--auth`,
- `authenticationMechanisms` parameter set to `PLAIN`, and
- `saslauthdPath` parameter set to the path to the Unix-domain Socket of the `saslauthd` instance.

Configure the MongoDB server using either the command line option `--setParameter` or the `configuration file`. Specify additional configurations as appropriate for your configuration.

If you use the `authorization` option to enforce authentication, you will need privileges to create a user.

Use specific `saslauthd` socket path. For socket path of `/<some>/<path>/saslauthd`, set the `saslauthdPath` to `/<some>/<path>/saslauthd/mux`, as in the following command line example:

```
mongod --auth --setParameter saslauthdPath=/<some>/<path>/saslauthd/mux --setParameter authentication
```

Or if using a `YAML format configuration file`, specify the following settings in the file:

```
security:
   authorization: enabled

setParameter:
   saslauthdPath: /<some>/<path>/saslauthd/mux
   authenticationMechanisms: PLAIN
```

Or, if using the older configuration file format[57]:

```
auth=true
setParameter=saslauthdPath=/<some>/<path>/saslauthd/mux
setParameter=authenticationMechanisms=PLAIN
```

Use default Unix-domain socket path. To use the default Unix-domain socket path, set the `saslauthdPath` to the empty string `""`, as in the following command line example:

```
mongod --auth --setParameter saslauthdPath="" --setParameter authenticationMechanisms=PLAIN
```

Or if using a `YAML format configuration file`, specify the following settings in the file:

```
security:
   authorization: enabled

setParameter:
   saslauthdPath: ""
   authenticationMechanisms: PLAIN
```

Or, if using the older configuration file format[58]:

```
auth=true
setParameter=saslauthdPath=""
setParameter=authenticationMechanisms=PLAIN
```

[57] http://docs.mongodb.org/v2.4/reference/configuration-options
[58] http://docs.mongodb.org/v2.4/reference/configuration-options

Step 3: Authenticate the user in the mongo shell. To perform the authentication in the mongo shell, use the db.auth() method in the $external database.

Specify the value "PLAIN" in the mechanism field, the user and password in the user and pwd fields respectively, and the value false in the digestPassword field. You **must** specify false for digestPassword since the server must receive an undigested password to forward on to saslauthd, as in the following example:

```
db.getSiblingDB("$external").auth(
   {
     mechanism: "PLAIN",
     user: <username>,
     pwd:  <cleartext password>,
     digestPassword: false
   }
)
```

The server forwards the password in plain text. In general, use only on a trusted channel (VPN, TLS/SSL, trusted wired network). See Considerations.

Configure MongoDB with Kerberos Authentication on Linux

On this page

New in version 2.4.

Overview

MongoDB Enterprise supports authentication using a *Kerberos service* (page 326). Kerberos is an industry standard authentication protocol for large client/server system.

Prerequisites

Setting up and configuring a Kerberos deployment is beyond the scope of this document. This tutorial assumes you have have configured a *Kerberos service principal* (page 327) for each mongod and mongos instance in your MongoDB deployment, and you have a valid *keytab file* (page 328) for for each mongod and mongos instance.

To verify MongoDB Enterprise binaries:

```
mongod --version
```

In the output from this command, look for the string modules: subscription or modules: enterprise to confirm your system has MongoDB Enterprise.

Procedure

The following procedure outlines the steps to add a Kerberos user principal to MongoDB, configure a standalone mongod instance for Kerberos support, and connect using the mongo shell and authenticate the user principal.

Step 1: Start `mongod` without Kerberos. For the initial addition of Kerberos users, start `mongod` without Kerberos support.

If a Kerberos user is already in MongoDB and has the *privileges required to create a user*, you can start `mongod` with Kerberos support.

Step 2: Connect to `mongod`. Connect via the `mongo` shell to the `mongod` instance. If `mongod` has `--auth` enabled, ensure you connect with the *privileges required to create a user*.

Step 3: Add Kerberos Principal(s) to MongoDB. Add a Kerberos principal, `<username>@<KERBEROS REALM>` or `<username>/<instance>@<KERBEROS REALM>`, to MongoDB in the `$external` database. Specify the Kerberos realm in all uppercase. The `$external` database allows `mongod` to consult an external source (e.g. Kerberos) to authenticate. To specify the user's privileges, assign *roles* (page 320) to the user.

The following example adds the Kerberos principal `application/reporting@EXAMPLE.NET` with read-only access to the `records` database:

```
use $external
db.createUser(
   {
     user: "application/reporting@EXAMPLE.NET",
     roles: [ { role: "read", db: "records" } ]
   }
)
```

Add additional principals as needed. For every user you want to authenticate using Kerberos, you must create a corresponding user in MongoDB. For more information about creating and managing users, see `http://docs.mongodb.org/manual/reference/command/nav-user-management`.

Step 4: Start `mongod` with Kerberos support. To start `mongod` with Kerberos support, set the environmental variable `KRB5_KTNAME` to the path of the keytab file and the `mongod` parameter `authenticationMechanisms` to `GSSAPI` in the following form:

```
env KRB5_KTNAME=<path to keytab file> \
mongod \
--setParameter authenticationMechanisms=GSSAPI
<additional mongod options>
```

For example, the following starts a standalone `mongod` instance with Kerberos support:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
/opt/mongodb/bin/mongod --auth \
--setParameter authenticationMechanisms=GSSAPI \
--dbpath /opt/mongodb/data
```

The path to your `mongod` as well as your *keytab file* (page 328) may differ. Modify or include additional `mongod` options as required for your configuration. The *keytab file* (page 328) must be only accessible to the owner of the `mongod` process.

With the official `.deb` or `.rpm` packages, you can set the `KRB5_KTNAME` in a environment settings file. See *KRB5_KTNAME* (page 371) for details.

Step 5: Connect `mongo` shell to `mongod` and authenticate. Connect the `mongo` shell client as the Kerberos principal `application/reporting@EXAMPLE.NET`. Before connecting, you must have used Kerberos's `kinit` program to get credentials for `application/reporting@EXAMPLE.NET`.

You can connect and authenticate from the command line.

```
mongo --authenticationMechanism=GSSAPI --authenticationDatabase='$external' \
--username application/reporting@EXAMPLE.NET
```

Or, alternatively, you can first connect `mongo` to the `mongod`, and then from the `mongo` shell, use the `db.auth()` method to authenticate in the `$external` database.

```
use $external
db.auth( { mechanism: "GSSAPI", user: "application/reporting@EXAMPLE.NET" } )
```

Additional Considerations

KRB5_KTNAME If you installed MongoDB Enterprise using one of the official `.deb` or `.rpm` packages, and you use the included init/upstart scripts to control the `mongod` instance, you can set the `KR5_KTNAME` variable in the default environment settings file instead of setting the variable each time.

For `.rpm` packages, the default environment settings file is `/etc/sysconfig/mongod`.

For `.deb` packages, the file is `/etc/default/mongodb`.

Set the `KRB5_KTNAME` value in a line that resembles the following:

```
export KRB5_KTNAME="<path to keytab>"
```

Configure `mongos` for Kerberos To start `mongos` with Kerberos support, set the environmental variable `KRB5_KTNAME` to the path of its *keytab file* (page 328) and the `mongos` parameter `authenticationMechanisms` to `GSSAPI` in the following form:

```
env KRB5_KTNAME=<path to keytab file> \
mongos \
--setParameter authenticationMechanisms=GSSAPI \
<additional mongos options>
```

For example, the following starts a `mongos` instance with Kerberos support:

```
env KRB5_KTNAME=/opt/mongodb/mongos.keytab \
mongos \
--setParameter authenticationMechanisms=GSSAPI \
--configdb shard0.example.net, shard1.example.net,shard2.example.net \
--keyFile /opt/mongodb/mongos.keyfile
```

The path to your `mongos` as well as your *keytab file* (page 328) may differ. The *keytab file* (page 328) must be only accessible to the owner of the `mongos` process.

Modify or include any additional `mongos` options as required for your configuration. For example, instead of using *`--keyFile`* for internal authentication of sharded cluster members, you can use *x.509 member authentication* (page 359) instead.

Use a Config File To configure `mongod` or `mongos` for Kerberos support using a `configuration file`, specify the `authenticationMechanisms` setting in the configuration file:

If using the `YAML configuration file format`:

```
setParameter:
   authenticationMechanisms: GSSAPI
```

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

```
setParameter=authenticationMechanisms=GSSAPI
```

Modify or include any additional `mongod` options as required for your configuration. For example, if `/opt/mongodb/mongod.conf` contains the following configuration settings for a standalone `mongod`:

```
security:
   authorization: enabled
setParameter:
   authenticationMechanisms: GSSAPI
storage:
   dbPath: /opt/mongodb/data
```

Or, if using the older configuration file format[59]:

```
auth = true
setParameter=authenticationMechanisms=GSSAPI
dbpath=/opt/mongodb/data
```

To start `mongod` with Kerberos support, use the following form:

```
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
/opt/mongodb/bin/mongod --config /opt/mongodb/mongod.conf
```

The path to your `mongod`, *keytab file* (page 328), and configuration file may differ. The *keytab file* (page 328) must be only accessible to the owner of the `mongod` process.

Troubleshoot Kerberos Setup for MongoDB If you encounter problems when starting `mongod` or `mongos` with Kerberos authentication, see *Troubleshoot Kerberos Authentication on Linux* (page 377).

Incorporate Additional Authentication Mechanisms Kerberos authentication (`GSSAPI`) can work alongside MongoDB's challenge/response authentication mechanism (`MONGODB-CR`), MongoDB's authentication mechanism for LDAP (`PLAIN`), and MongoDB's authentication mechanism for x.509 (`MONGODB-X509`). Specify the mechanisms, as follows:

```
--setParameter authenticationMechanisms=GSSAPI,MONGODB-CR
```

Only add the other mechanisms if in use. This parameter setting does not affect MongoDB's internal authentication of cluster members.

Additional Resources

- MongoDB LDAP and Kerberos Authentication with Dell (Quest) Authentication Services[60]
- MongoDB with Red Hat Enterprise Linux Identity Management and Kerberos[61]

Configure MongoDB with Kerberos Authentication on Windows

[59] http://docs.mongodb.org/v2.4/reference/configuration-options
[60] https://www.mongodb.com/blog/post/mongodb-ldap-and-kerberos-authentication-dell-quest-authentication-services?jmp=docs
[61] http://docs.mongodb.org/ecosystem/tutorial/manage-red-hat-enterprise-linux-identity-management/?jmp=docs

On this page

New in version 2.6.

Overview

MongoDB Enterprise supports authentication using a *Kerberos service* (page 326). Kerberos is an industry standard authentication protocol for large client/server system. Kerberos allows MongoDB and applications to take advantage of existing authentication infrastructure and processes.

Prerequisites

Setting up and configuring a Kerberos deployment is beyond the scope of this document. This tutorial assumes have configured a *Kerberos service principal* (page 327) for each `mongod.exe` and `mongos.exe` instance.

Procedures

Step 1: Start `mongod.exe` without Kerberos. For the initial addition of Kerberos users, start `mongod.exe` without Kerberos support.

If a Kerberos user is already in MongoDB and has the *privileges required to create a user*, you can start `mongod.exe` with Kerberos support.

Step 2: Connect to `mongod`. Connect via the `mongo.exe` shell to the `mongod.exe` instance. If `mongod.exe` has `--auth` enabled, ensure you connect with the *privileges required to create a user*.

Step 3: Add Kerberos Principal(s) to MongoDB. Add a Kerberos principal, `<username>@<KERBEROS REALM>`, to MongoDB in the `$external` database. Specify the Kerberos realm in **ALL UPPERCASE**. The `$external` database allows `mongod.exe` to consult an external source (e.g. Kerberos) to authenticate. To specify the user's privileges, assign *roles* (page 320) to the user.

The following example adds the Kerberos principal `reportingapp@EXAMPLE.NET` with read-only access to the `records` database:

```
use $external
db.createUser(
   {
     user: "reportingapp@EXAMPLE.NET",
     roles: [ { role: "read", db: "records" } ]
   }
)
```

Add additional principals as needed. For every user you want to authenticate using Kerberos, you must create a corresponding user in MongoDB. For more information about creating and managing users, see `http://docs.mongodb.org/manual/reference/command/nav-user-management`.

Step 4: Start `mongod.exe` with Kerberos support. You must start `mongod.exe` as the *service principal account* (page 374).

To start `mongod.exe` with Kerberos support, set the `mongod.exe` parameter `authenticationMechanisms` to `GSSAPI`:

```
mongod.exe --setParameter authenticationMechanisms=GSSAPI <additional mongod.exe options>
```

For example, the following starts a standalone `mongod.exe` instance with Kerberos support:

```
mongod.exe --auth --setParameter authenticationMechanisms=GSSAPI
```

Modify or include additional `mongod.exe` options as required for your configuration.

Step 5: Connect `mongo.exe` shell to `mongod.exe` and authenticate. Connect the `mongo.exe` shell client as the Kerberos principal `application@EXAMPLE.NET`.

You can connect and authenticate from the command line.

```
mongo.exe --authenticationMechanism=GSSAPI --authenticationDatabase='$external' \
--username reportingapp@EXAMPLE.NET
```

Or, alternatively, you can first connect `mongo.exe` to the `mongod.exe`, and then from the `mongo.exe` shell, use the `db.auth()` method to authenticate in the `$external` database.

```
use $external
db.auth( { mechanism: "GSSAPI", user: "reportingapp@EXAMPLE.NET" } )
```

Additional Considerations

Configure `mongos.exe` for Kerberos To start `mongos.exe` with Kerberos support, set the `mongos.exe` parameter `authenticationMechanisms` to `GSSAPI`. You must start `mongos.exe` as the *service principal account* (page 374).:

```
mongos.exe --setParameter authenticationMechanisms=GSSAPI <additional mongos options>
```

For example, the following starts a `mongos` instance with Kerberos support:

```
mongos.exe --setParameter authenticationMechanisms=GSSAPI --configdb shard0.example.net, shard1.examp
```

Modify or include any additional `mongos.exe` options as required for your configuration. For example, instead of using *`--keyFile`* for for internal authentication of sharded cluster members, you can use *x.509 member authentication* (page 359) instead.

Assign Service Principal Name to MongoDB Windows Service Use `setspn.exe` to assign the service principal name (SPN) to the account running the `mongod.exe` and the `mongos.exe` service:

```
setspn.exe -A <service>/<fully qualified domain name> <service account name>
```

For example, if `mongod.exe` runs as a service named `mongodb` on `testserver.mongodb.com` with the service account name `mongodtest`, assign the SPN as follows:

```
setspn.exe -A mongodb/testserver.mongodb.com mongodtest
```

Incorporate Additional Authentication Mechanisms Kerberos authentication (`GSSAPI`) can work alongside MongoDB's challenge/response authentication mechanism (`MONGODB-CR`), MongoDB's authentication mechanism for LDAP (`PLAIN`), and MongoDB's authentication mechanism for x.509 (`MONGODB-X509`). Specify the mechanisms, as follows:

```
--setParameter authenticationMechanisms=GSSAPI,MONGODB-CR
```

Only add the other mechanisms if in use. This parameter setting does not affect MongoDB's internal authentication of cluster members.

Authenticate to a MongoDB Instance or Cluster

On this page

Overview

To authenticate to a running `mongod` or `mongos` instance, you must have user credentials for a resource on that instance. When you authenticate to MongoDB, you authenticate either to a database or to a cluster. Your user privileges determine the resource you can authenticate to.

You authenticate to a resource either by:

- using the authentication options when connecting to the `mongod` or `mongos` instance, or
- connecting first and then authenticating to the resource with the `authenticate` command or the `db.auth()` method.

This section describes both approaches.

In general, always use a trusted channel (VPN, TLS/SSL, trusted wired network) for connecting to a MongoDB instance.

Prerequisites

You must have user credentials on the database or cluster to which you are authenticating.

Procedures

Authenticate When First Connecting to MongoDB

Step 1: Specify your credentials when starting the `mongo` instance. When using `mongo` to connect to a `mongod` or `mongos`, enter your `username`, `password`, and *`authenticationDatabase`*. For example:

```
mongo --username "prodManager" --password "cleartextPassword" --authenticationDatabase "products"
```

Step 2: Close the session when your work is complete. To close an authenticated session, use the `logout` command.:

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

Authenticate After Connecting to MongoDB

Step 1: Connect to a MongoDB instance. Connect to a `mongod` or `mongos` instance.

Step 2: Switch to the database to which to authenticate.

```
use <database>
```

Step 3: Authenticate. Use either the `authenticate` command or the `db.auth()` method to provide your username and password to the database. For example:

```
db.auth( "prodManager", "cleartextPassword" )
```

Step 4: Close the session when your work is complete. To close an authenticated session, use the `logout` command.:

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

Generate a Key File

On this page

Overview

This section describes how to generate a key file to store authentication information. After generating a key file, specify the key file using the `keyFile` option when starting a `mongod` or `mongos` instance.

A key's length must be between 6 and 1024 characters and may only contain characters in the base64 set. The key file must not have group or world permissions on UNIX systems. Key file permissions are not checked on Windows systems.

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
```

Procedure

Step 1: Create a key file. Create the key file your deployment will use to authenticate servers to each other.

To generate pseudo-random data to use for a `keyfile`, issue the following `openssl` command:

```
openssl rand -base64 741 > mongodb-keyfile
chmod 600 mongodb-keyfile
```

You may generate a key file using any method you choose. Always ensure that the password stored in the key file is both long and contains a high amount of entropy. Using `openssl` in this manner helps generate such a key.

Step 2: Specify the key file when starting a MongoDB instance. Specify the path to the key file with the `keyFile` option.

Troubleshoot Kerberos Authentication on Linux

On this page

New in version 2.4.

Kerberos Configuration Checklist

If you have difficulty starting `mongod` or `mongos` with *Kerberos* (page 326) on Linux systems, ensure that:

- The `mongod` and the `mongos` binaries are from MongoDB Enterprise.

  To verify MongoDB Enterprise binaries:

  ```
  mongod --version
  ```

  In the output from this command, look for the string `modules: subscription` or `modules: enterprise` to confirm your system has MongoDB Enterprise.

- You are not using the HTTP Console[62]. MongoDB Enterprise does not support Kerberos authentication over the HTTP Console interface.

- Either the service principal name (SPN) in the *keytab file* (page 328) matches the SPN for the `mongod` or `mongos` instance, or the `mongod` or the `mongos` instance use the `--setParameter saslHostName=<host name>` to match the name in the keytab file.

- The canonical system hostname of the system that runs the `mongod` or `mongos` instance is a resolvable, fully qualified domain for this host. You can test the system hostname resolution with the `hostname -f` command at the system prompt.

- Each host that runs a `mongod` or `mongos` instance has both the `A` and `PTR` DNS records to provide forward and reverse lookup. The records allow the host to resolve the components of the Kerberos infrastructure.

[62] https://docs.mongodb.org/ecosystem/tools/http-interface/#http-console

- Both the Kerberos Key Distribution Center (KDC) and the system running `mongod` instance or `mongos` must be able to resolve each other using DNS. By default, Kerberos attempts to resolve hosts using the content of the `/etc/kerb5.conf` before using DNS to resolve hosts.
- The time synchronization of the systems running `mongod` or the `mongos` instances and the Kerberos infrastructure are within the maximum time skew (default is 5 minutes) of each other. Time differences greater than the maximum time skew will prevent successful authentication.

Debug with More Verbose Logs

If you still encounter problems with Kerberos on Linux, you can start both `mongod` and `mongo` (or another client) with the environment variable `KRB5_TRACE` set to different files to produce more verbose logging of the Kerberos process to help further troubleshooting. For example, the following starts a standalone `mongod` with `KRB5_TRACE` set:

```
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. Some common error messages are:

`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 are using the correct credentials and the correct fully qualified domain name when connecting to the host.

`GSSAPI error acquiring credentials.` This error occurs during the start of the `mongod` or `mongos` and reflects improper configuration of the system hostname or a missing or incorrectly configured keytab file.

If you encounter this problem, consider the items in the *Kerberos Configuration Checklist* (page 377), in particular, whether the SPN in the *keytab file* (page 328) matches the SPN for the `mongod` or `mongos` instance.

To determine whether the SPNs match:

1. Examine the keytab file, with the following command:

   ```
   klist -k <keytab>
   ```

   Replace `<keytab>` with the path to your keytab file.

2. 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 start `mongod` or `mongos` with the `--setParameter saslHostName=<hostname>`.

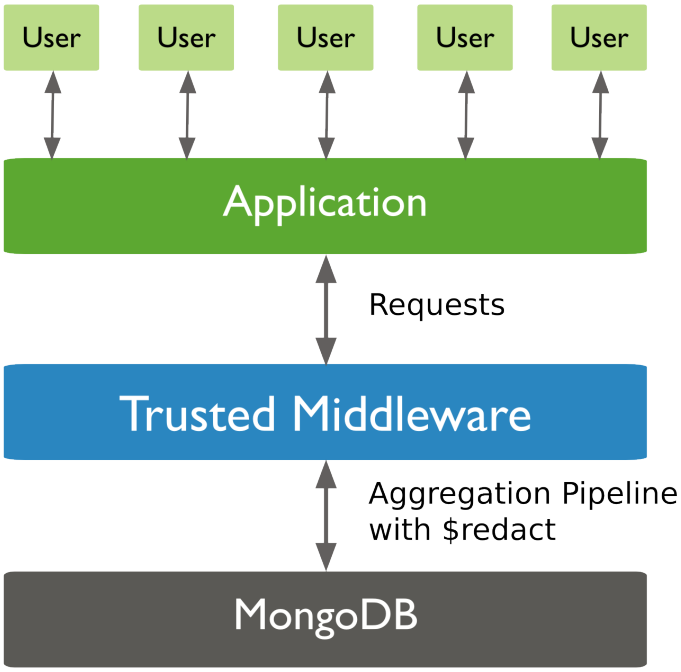
See also:

- *Kerberos Authentication* (page 326)
- *Configure MongoDB with Kerberos Authentication on Linux* (page 369)
- *Configure MongoDB with Kerberos Authentication on Windows* (page 372)

Implement Field Level Redaction

On this page

The `$redact` pipeline operator restricts the contents of the documents based on information stored in the documents themselves.



To store the access criteria data, add a field to the documents and embedded documents. To allow for multiple combinations of access levels for the same data, consider setting the access field to an array of arrays. Each array element contains a required set that allows a user with that set to access the data.

Then, include the `$redact` stage in the `db.collection.aggregate()` operation to restrict contents of the result set based on the access required to view the data.

For more information on the `$redact` pipeline operator, including its syntax and associated system variables as well as additional examples, see `$redact`.

Procedure

For example, a `forecasts` collection contains documents of the following form where the `tags` field determines the access levels required to view the data:

```
{
   _id: 1,
   title: "123 Department Report",
```

```
   tags: [ [ "G" ], [ "FDW" ] ],
   year: 2014,
   subsections: [
       {
           subtitle: "Section 1: Overview",
           tags: [ [ "SI", "G" ], [ "FDW" ] ],
           content:  "Section 1: This is the content of section 1."
       },
       {
           subtitle: "Section 2: Analysis",
           tags: [ [ "STLW" ] ],
           content: "Section 2: This is the content of section 2."
       },
       {
           subtitle: "Section 3: Budgeting",
           tags: [ [ "TK" ], [ "FDW", "TGE" ] ],
           content: {
               text: "Section 3: This is the content of section3.",
               tags: [ [ "HCS"], [ "FDW", "TGE", "BX" ] ]
           }
       }
   ]
}
```

For each document, the `tags` field contains various access groupings necessary to view the data. For example, the value `[ [ "G" ], [ "FDW", "TGE" ] ]` can specify that a user requires either access level `["G"]` or both `[ "FDW", "TGE" ]` to view the data.

Consider a user who only has access to view information tagged with either `"FDW"` or `"TGE"`. To run a query on all documents with year `2014` for this user, include a `$redact` stage as in the following:

```
var userAccess = [ "FDW", "TGE" ];
db.forecasts.aggregate(
   [
     { $match: { year: 2014 } },
     { $redact:
        {
          $cond: {
                   if: { $anyElementTrue:
                          {
                            $map: {
                                     input: "$tags" ,
                                     as: "fieldTag",
                                     in: { $setIsSubset: [ "$$fieldTag", userAccess ] }
                                   }
                          }
                      },
                    then: "$$DESCEND",
                    else: "$$PRUNE"
                  }
        }
     }
   ]
)
```

The aggregation operation returns the following "redacted" document for the user:

```
{ "_id" : 1,
  "title" : "123 Department Report",
```

```
  "tags" : [ [ "G" ], [ "FDW" ] ],
  "year" : 2014,
  "subsections" :
     [
        {
          "subtitle" : "Section 1: Overview",
          "tags" : [ [ "SI", "G" ], [ "FDW" ] ],
          "content" : "Section 1: This is the content of section 1."
        },
       {
         "subtitle" : "Section 3: Budgeting",
         "tags" : [ [ "TK" ], [ "FDW", "TGE" ] ]
       }
     ]
}
```

See also:

`$map`, `$setIsSubset`, `$anyElementTrue`

6.3.4 User and Role Management Tutorials

The following tutorials provide instructions on how to enable authentication and limit access for users with privilege roles.

Create a User Administrator **(page 381)** Create users with special permissions to to create, modify, and remove other users, as well as administer authentication credentials (e.g. passwords).

Add a User to a Database **(page 383)** Create non-administrator users using MongoDB's role-based authentication system.

Create an Administrative User with Unrestricted Access **(page 385)** Create a user with unrestricted access. Create such a user only in unique situations. In general, all users in the system should have no more access than needed to perform their required operations.

Create a Role **(page 386)** Create custom role.

Assign a User a Role **(page 388)** Assign a user a role. A role grants the user a defined set of privileges. A user can have multiple roles.

Verify User Privileges **(page 389)** View a user's current privileges.

Modify a User's Access **(page 391)** Modify the actions available to a user on specific database resources.

View Roles **(page 393)** View a role's privileges.

Change a User's Password **(page 394)** Only user administrators can edit credentials. This tutorial describes the process for editing an existing user's password.

Change Your Password and Custom Data **(page 395)** Users with sufficient access can change their own passwords and modify the optional *custom data* associated with their user credential.

Create a User Administrator

On this page

Overview

User administrators create users and create and assigns roles. A user administrator can grant any privilege in the database and can create new ones. In a MongoDB deployment, create the user administrator as the first user. Then let this user create all other users.

To provide user administrators, MongoDB has `userAdmin` (page 407) and `userAdminAnyDatabase` (page 411) roles, which grant access to *actions* (page 418) that support user and role management. Following the policy of *least privilege* `userAdmin` (page 407) and `userAdminAnyDatabase` (page 411) confer no additional privileges.

Carefully control access to these roles. A user with either of these roles can grant *itself* unlimited additional privileges. Specifically, a user with the `userAdmin` (page 407) role can grant itself any privilege in the database. A user assigned either the `userAdmin` (page 407) role on the `admin` database or the `userAdminAnyDatabase` (page 411) can grant itself any privilege *in the system*.

Prerequisites

Required Access You must have the `createUser` (page 420) *action* (page 418) on a database to create a new user on that database.

You must have the `grantRole` (page 420) *action* (page 418) on a role's database to grant the role to another user.

If you have the `userAdmin` (page 407) or `userAdminAnyDatabase` (page 411) role, you have those actions.

First User Restrictions If your MongoDB deployment has no users, you *must* connect to `mongod` using the *localhost exception* (page 319) or use the `--noauth` option when starting `mongod` to gain full access the system. Once you have access, you can skip to *Creating the system user administrator* in this procedure.

If users exist in the MongoDB database, but none of them has the appropriate prerequisites to create a new user or you do not have access to them, you *must* restart `mongod` with the `--noauth` option.

Procedure

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` either through the *localhost exception* (page 319) or as a user with the privileges indicated in the prerequisites section.

In the following example, `manager` has the required privileges specified in *Prerequisites* (page 382).

```
mongo --port 27017 -u manager -p 123456 --authenticationDatabase admin
```

Step 2: Create the system user administrator. Add the user with the `userAdminAnyDatabase` (page 411) role, and only that role.

The following example creates the user `siteUserAdmin` user on the `admin` database:

```
use admin
db.createUser(
  {
    user: "siteUserAdmin",
    pwd: "password",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

Step 3: Create a user administrator for a single database. Optionally, you may want to create user administrators that only have access to administer users in a specific database by way of the `userAdmin` (page 407) role.

The following example creates the user `recordsUserAdmin` on the `records` database:

```
use records
db.createUser(
  {
    user: "recordsUserAdmin",
    pwd: "password",
    roles: [ { role: "userAdmin", db: "records" } ]
  }
)
```

Related Documents

- *Authentication* (page 316)
- *Security Introduction* (page 313)
- *Enable Client Access Control* (page 353)
- *Access Control Tutorials* (page 352)

Additional Resources

- Security Architecture White Paper[63]
- Webinar: Securing Your MongoDB Deployment[64]
- Creating a Single View Part 3: Securing Your Deployment[65]

Add a User to a Database

On this page

Changed in version 2.6.

[63] https://www.mongodb.com/lp/white-paper/mongodb-security-architecture?jmp=docs
[64] http://www.mongodb.com/webinar/securing-your-mongodb-deployment?jmp=docs
[65] https://www.mongodb.com/presentations/creating-single-view-part-3-securing-your-deployment?jmp=docs

Overview

Each application and user of a MongoDB system should map to a distinct application or administrator. This *access isolation* facilitates access revocation and ongoing user maintenance. At the same time users should have only the minimal set of privileges required to ensure a system of *least privilege*.

To create a user, you must define the user's credentials and assign that user *roles* (page 320). Credentials verify the user's identity to a database, and roles determine the user's access to database resources and operations.

For an overview of credentials and roles in MongoDB see *Security Introduction* (page 313).

Considerations

For users that authenticate using external mechanisms, [66] you do not need to provide credentials when creating users.

For all users, select the roles that have the exact required *privileges* (page 320). If the correct roles do not exist, *create roles* (page 386).

You can create a user without assigning roles, choosing instead to assign the roles later. To do so, create the user with an empty `roles` (page 416) array.

Prerequisites

To create a user on a system that uses *authentication* (page 316), you must authenticate as a user administrator. If you have not yet created a user administrator, do so as described in *Create a User Administrator* (page 381).

Required Access You must have the `createUser` (page 420) *action* (page 418) on a database to create a new user on that database.

You must have the `grantRole` (page 420) *action* (page 418) on a role's database to grant the role to another user.

If you have the `userAdmin` (page 407) or `userAdminAnyDatabase` (page 411) role, you have those actions.

First User Restrictions If your MongoDB deployment has no users, you *must* connect to `mongod` using the *localhost exception* (page 319) or use the `--noauth` option when starting `mongod` to gain full access the system. Once you have access, you can skip to *Creating the system user administrator* in this procedure.

If users exist in the MongoDB database, but none of them has the appropriate prerequisites to create a new user or you do not have access to them, you *must* restart `mongod` with the `--noauth` option.

Procedures

Step 1: Connect to MongoDB with the appropriate privileges. Connect to the `mongod` or `mongos` with the privileges specified in the *Prerequisites* (page 384) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

[66] *Configure MongoDB with Kerberos Authentication on Linux* (page 369), *Authenticate Using SASL and LDAP with OpenLDAP* (page 366), *Authenticate Using SASL and LDAP with ActiveDirectory* (page 363), and x.509 certificates provide external authentication mechanisms.

Step 2: Create the new user. Create the user in the database to which the user will belong. Pass a well formed user document to the `db.createUser()` method.

The following operation creates a user in the `reporting` database with the specified name, password, and roles.

```
use reporting
db.createUser(
    {
      user: "reportsUser",
      pwd: "12345678",
      roles: [
         { role: "read", db: "reporting" },
         { role: "read", db: "products" },
         { role: "read", db: "sales" },
         { role: "readWrite", db: "accounts" }
      ]
    }
)
```

To authenticate the `reportsUser`, you must authenticate the user in the `reporting` database.

Create an Administrative User with Unrestricted Access

On this page

Overview

Most users should have only the minimal set of privileges required for their operations, in keeping with the policy of *least privilege*. However, some authorization architectures may require a user with unrestricted access. To support these *super users*, you can create users with access to all database *resources* (page 417) and *actions* (page 418).

For many deployments, you may be able to avoid having *any* users with unrestricted access by having an administrative user with the `createUser` (page 420) and `grantRole` (page 420) actions granted as needed to support operations.

If users truly need unrestricted access to a MongoDB deployment, MongoDB provides a *built-in role* (page 405) named `root` (page 412) that grants the combined privileges of all built-in roles. This document describes how to create an administrative user with the `root` (page 412) role.

For descriptions of the access each built-in role provides, see the section on *built-in roles* (page 405).

Prerequisites

Required Access You must have the `createUser` (page 420) *action* (page 418) on a database to create a new user on that database.

You must have the `grantRole` (page 420) *action* (page 418) on a role's database to grant the role to another user.

If you have the `userAdmin` (page 407) or `userAdminAnyDatabase` (page 411) role, you have those actions.

First User Restrictions If your MongoDB deployment has no users, you *must* connect to `mongod` using the *localhost exception* (page 319) or use the `--noauth` option when starting `mongod` to gain full access the system. Once you have access, you can skip to *Creating the system user administrator* in this procedure.

If users exist in the MongoDB database, but none of them has the appropriate prerequisites to create a new user or you do not have access to them, you *must* restart `mongod` with the `--noauth` option.

Procedure

Step 1: Connect to MongoDB with the appropriate privileges. Connect to the `mongod` or `mongos` as a user with the privileges specified in the *Prerequisites* (page 385) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Create the administrative user. In the `admin` database, create a new user using the `db.createUser()` method. Give the user the built-in `root` (page 412) role.

For example:

```
use admin
db.createUser(
    {
      user: "superuser",
      pwd: "12345678",
      roles: [ "root" ]
    }
)
```

Authenticate against the `admin` database to test the new user account. Use `db.auth()` while using the `admin` database or use the `mongo` shell with the `--authenticationDatabase` option.

Create a Role

On this page

Overview

Roles grant users access to MongoDB resources. By default, MongoDB provides a number of *built-in roles* (page 405) that administrators may use to control access to a MongoDB system. However, if these roles cannot describe the desired set of privileges, you can create a new, customized role in a particular database.

Except for roles created in the `admin` database, a role can only include privileges that apply to its database and can only inherit from other roles in its database.

A role created in the `admin` database can include privileges that apply to the `admin` database, other databases or to the *cluster* (page 418) resource, and can inherit from roles in other databases as well as the `admin` database.

MongoDB uses the combination of the database name and the role name to uniquely define a role.

Prerequisites

To create a role in a database, the user must have:

- the `createRole` (page 420) *action* (page 418) on that *database resource* (page 417).
- the `grantRole` (page 420) *action* (page 418) on that database to specify privileges for the new role as well as to specify roles to inherit from.

Built-in roles `userAdmin` (page 407) and `userAdminAnyDatabase` (page 411) provide `createRole` (page 420) and `grantRole` (page 420) actions on their respective *resources* (page 417).

Procedures

To create a new role, use the `db.createRole()` method, specifying the privileges in the `privileges` array and the inherited roles in the `roles` array.

Create a Role to Manage Current Operations The following example creates a role named `manageOpRole` which provides only the privileges to run both `db.currentOp()` and `db.killOp()`. [67]

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` with the privileges specified in the *Prerequisites* (page 387) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

The `siteUserAdmin` has privileges to create roles in the `admin` as well as other databases.

Step 2: Create a new role to manage current operations. `manageOpRole` has privileges that act on multiple databases as well as the *cluster resource* (page 418). As such, you must create the role in the `admin` database.

```
use admin
db.createRole(
   {
     role: "manageOpRole",
     privileges: [
       { resource: { cluster: true }, actions: [ "killop", "inprog" ] },
       { resource: { db: "", collection: "" }, actions: [ "killCursors" ] }
     ],
     roles: []
   }
)
```

The new role grants permissions to kill any operations.

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

Create a Role to Run `mongostat` The following example creates a role named `mongostatRole` that provides only the privileges to run `mongostat`. [68]

[67] The built-in role `clusterMonitor` (page 408) also provides the privilege to run `db.currentOp()` along with other privileges, and the built-in role `hostManager` (page 409) provides the privilege to run `db.killOp()` along with other privileges.

[68] The built-in role `clusterMonitor` (page 408) also provides the privilege to run `mongostat` along with other privileges.

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` with the privileges specified in the *Prerequisites* (page 387) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

The `siteUserAdmin` has privileges to create roles in the `admin` as well as other databases.

Step 2: Create a new role to manage current operations. `mongostatRole` has privileges that act on the *cluster resource* (page 418). As such, you must create the role in the `admin` database.

```
use admin
db.createRole(
   {
     role: "mongostatRole",
     privileges: [
       { resource: { cluster: true }, actions: [ "serverStatus" ] }
     ],
     roles: []
   }
)
```

Assign a User a Role

On this page

Changed in version 2.6.

Overview

A role provides a user privileges to perform a set of *actions* (page 418) on a *resource* (page 417). A user can have multiple roles.

In MongoDB systems with `authorization` enforced, you must grant a user a role for the user to access a database resource. To assign a role, first determine the privileges the user needs and then determine the role that grants those privileges.

For an overview of roles and privileges, see *Authorization* (page 320). For descriptions of the access each built-in role provides, see the section on *built-in roles* (page 405).

Prerequisites

You must have the `grantRole` (page 420) *action* (page 418) on a database to grant a role on that database.

To view a role's information, you must be explicitly granted the role or must have the `viewRole` (page 420) *action* (page 418) on the role's database.

Procedure

Step 1: Connect with the privilege to grant roles. Connect to the `mongod` or `mongos` as a user with the privileges specified in the *Prerequisites* (page 388) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Identify the user's roles and privileges. To display the roles and privileges of the user to be modified, use the `db.getUser()` and `db.getRole()` methods.

For example, to view roles for `reportsUser` created in *Add a User to a Database* (page 383), issue:

```
use reporting
db.getUser("reportsUser")
```

To display the privileges granted to the user by the `readWrite` role on the `"accounts"` database, issue:

```
use accounts
db.getRole( "readWrite", { showPrivileges: true } )
```

Step 3: Identify the privileges to grant or revoke. If the user requires additional privileges, grant to the user the role, or roles, with the required set of privileges. If such a role does not exist, *create a new role* (page 386) with the appropriate set of privileges.

Step 4: Grant a role to a user. Grant the user the role using the `db.grantRolesToUser()` method.

For example, the following grants new roles to the user `reportsUser` created in *Add a User to a Database* (page 383).

```
use reporting
db.grantRolesToUser(
  "reportsUser",
  [
    { role: "readWrite", db: "products" } ,
    { role: "readAnyDatabase", db:"admin" }
  ]
)
```

Verify User Privileges

On this page

Overview

A user's privileges determine the access the user has to MongoDB *resources* (page 417) and the *actions* (page 418) that user can perform. Users receive privileges through role assignments. A user can have multiple roles, and each

role can have multiple privileges.

For an overview of roles and privileges, see *Authorization* (page 320).

Prerequisites

To view a role's information, you must be explicitly granted the role or must have the `viewRole` (page 420) *action* (page 418) on the role's database.

Procedure

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` as a user with the privileges specified in the prerequisite section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Identify the user's roles. Use the `usersInfo` command or `db.getUser()` method to display user information.

For example, to view roles for `reportsUser` created in *Add a User to a Database* (page 383), issue:

```
use reporting
db.getUser("reportsUser")
```

In the returned document, the `roles` (page 416) field displays all roles for `reportsUser`:

```
...
"roles" : [
   { "role" : "readWrite", "db" : "accounts" },
   { "role" : "read", "db" : "reporting" },
   { "role" : "read", "db" : "products" },
   { "role" : "read", "db" : "sales" }
]
```

Step 3: Identify the privileges granted by the roles. For a given role, use the `db.getRole()` method, or the `rolesInfo` command, with the `showPrivileges` option:

For example, to view the privileges granted by `read` role on the `products` database, use the following operation, issue:

```
use products
db.getRole( "read", { showPrivileges: true } )
```

In the returned document, the `privileges` and `inheritedPrivileges` arrays. The `privileges` lists the privileges directly specified by the role and excludes those privileges inherited from other roles. The `inheritedPrivileges` lists all privileges granted by this role, both directly specified and inherited. If the role does not inherit from other roles, the two fields are the same.

```
...
"privileges" : [
  {
    "resource": { "db" : "products", "collection" : "" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
```

```
  {
    "resource" : { "db" : "products", "collection" : "system.indexes" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
  {
    "resource" : { "db" : "products", "collection" : "system.js" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
  {
    "resource" : { "db" : "products", "collection" : "system.namespaces" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  }
],
"inheritedPrivileges" : [
  {
    "resource": { "db" : "products", "collection" : "" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
  {
    "resource" : { "db" : "products", "collection" : "system.indexes" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
  {
    "resource" : { "db" : "products", "collection" : "system.js" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  },
  {
    "resource" : { "db" : "products", "collection" : "system.namespaces" },
    "actions": [ "collStats","dbHash","dbStats","find","killCursors","planCacheRead" ]
  }
]
```

Modify a User's Access

On this page

Overview

When a user's responsibilities change, modify the user's access to include only those roles the user requires. This follows the policy of *least privilege*.

To change a user's access, first determine the privileges the user needs and then determine the roles that grants those privileges. Grant and revoke roles using the `db.grantRolesToUser()` and `db.revokeRolesFromUser()` methods.

For an overview of roles and privileges, see *Authorization* (page 320). For descriptions of the access each built-in role provides, see the section on *built-in roles* (page 405).

Prerequisites

You must have the `grantRole` (page 420) *action* (page 418) on a database to grant a role on that database.

You must have the `revokeRole` (page 420) *action* (page 418) on a database to revoke a role on that database.

To view a role's information, you must be explicitly granted the role or must have the `viewRole` (page 420) *action* (page 418) on the role's database.

Procedure

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` as a user with the privileges specified in the prerequisite section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Identify the user's roles and privileges. To display the roles and privileges of the user to be modified, use the `db.getUser()` and `db.getRole()` methods.

For example, to view roles for `reportsUser` created in *Add a User to a Database* (page 383), issue:

```
use reporting
db.getUser("reportsUser")
```

To display the privileges granted to the user by the `readWrite` role on the `"accounts"` database, issue:

```
use accounts
db.getRole( "readWrite", { showPrivileges: true } )
```

Step 3: Identify the privileges to grant or revoke. If the user requires additional privileges, grant to the user the role, or roles, with the required set of privileges. If such a role does not exist, *create a new role* (page 386) with the appropriate set of privileges.

To revoke a subset of privileges provided by an existing role: revoke the original role and grant a role that contains only the required privileges. You may need to *create a new role* (page 386) if a role does not exist.

Step 4: Modify the user's access.

Revoke a Role Revoke a role with the `db.revokeRolesFromUser()` method. The following example operation removes the `readWrite` (page 405) role on the `accounts` database from the `reportsUser`:

```
use reporting
db.revokeRolesFromUser(
    "reportsUser",
    [
      { role: "readWrite", db: "accounts" }
    ]
)
```

Grant a Role Grant a role using the `db.grantRolesToUser()` method. For example, the following operation grants the `reportsUser` user the `read` (page 405) role on the `accounts` database:

```
use reporting
db.grantRolesToUser(
    "reportsUser",
    [
      { role: "read", db: "accounts" }
    ]
)
```

For sharded clusters, the changes to the user are instant on the `mongos` on which the command runs. However, for other `mongos` instances in the cluster, the user cache may wait up to 10 minutes to refresh. See `userCacheInvalidationIntervalSecs`.

View Roles

On this page

Overview

A *role* (page 320) grants privileges to the users who are assigned the role. Each role is scoped to a particular database, but MongoDB stores all role information in the `admin.system.roles` (page 304) collection in the `admin` database.

Prerequisites

To view a role's information, you must be explicitly granted the role or must have the `viewRole` (page 420) *action* (page 418) on the role's database.

Procedures

The following procedures use the `rolesInfo` command. You also can use the methods `db.getRole()` (singular) and `db.getRoles()`.

View a Role in the Current Database If the role is in the current database, you can refer to the role by name, as for the role `dataEntry` on the current database:

```
db.runCommand({ rolesInfo: "dataEntry" })
```

View a Role in a Different Database If the role is in a different database, specify the role as a document. Use the following form:

```
{ role: "<role name>", db: "<role db>" }
```

To view the custom `appWriter` role in the `orders` database, issue the following command from the `mongo` shell:

```
db.runCommand({ rolesInfo: { role: "appWriter", db: "orders" } })
```

View Multiple Roles To view information for multiple roles, specify each role as a document or string in an array.

To view the custom `appWriter` and `clientWriter` roles in the `orders` database, as well as the `dataEntry` role on the current database, use the following command from the `mongo` shell:

```
db.runCommand( { rolesInfo: [ { role: "appWriter", db: "orders" },
                              { role: "clientWriter", db: "orders" },
                              "dataEntry" ]
               } )
```

View All Custom Roles To view the all custom roles, query *admin.system.roles* (page 413) collection directly, for example:

```
db = db.getSiblingDB('admin')
db.system.roles.find()
```

Change a User's Password

Changed in version 2.6.

On this page

Overview

Strong passwords help prevent unauthorized access, and all users should have strong passwords. You can use the `openssl` program to generate unique strings for use in passwords, as in the following command:

```
openssl rand -base64 48
```

Prerequisites

You must have the `changeAnyPassword` *action* (page 418) on a database to modify the password of any user on that database.

To change your own password, you must have the `changeOwnPassword` (page 419) *action* (page 418) on your database. See *Change Your Password and Custom Data* (page 395).

Procedure

Step 1: Connect to MongoDB with the appropriate privileges. Connect to the `mongod` or `mongos` with the privileges specified in the *Prerequisites* (page 394) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Change the password. Pass the user's username and the new password to the `db.changeUserPassword()` method.

The following operation changes the `reporting` user's password to `SOh3TbYhxuLiW8ypJPxmt1oOfL`:

```
db.changeUserPassword("reporting", "SOh3TbYhxuLiW8ypJPxmt1oOfL")
```

Change Your Password and Custom Data

Changed in version 2.6.

On this page

Overview

Users with appropriate privileges can change their own passwords and custom data. `Custom data` (page 416) stores optional user information.

Considerations

To generate a strong password for use in this procedure, you can use the `openssl` utility's `rand` command. For example, issue `openssl rand` with the following options to create a base64-encoded string of 48 pseudo-random bytes:

```
openssl rand -base64 48
```

Prerequisites

To modify your own password and custom data, you must have privileges that grant `changeOwnPassword` (page 419) and `changeOwnCustomData` (page 419) *actions* (page 418) respectively on the user's database.

Step 1: Connect as a user with privileges to manage users and roles. Connect to the `mongod` or `mongos` with privileges to manage users and roles, such as a user with `userAdminAnyDatabase` (page 411) role. The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 381).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

Step 2: Create a role with appropriate privileges. In the `admin` database, `create` a new role with `changeOwnPassword` (page 419) and `changeOwnCustomData` (page 419).

```
use admin
db.createRole(
   { role: "changeOwnPasswordCustomDataRole",
     privileges: [
        {
          resource: { db: "", collection: ""},
          actions: [ "changeOwnPassword", "changeOwnCustomData" ]
        }
     ],
     roles: []
   }
)
```

Step 3: Add a user with this role. In the `test` database, `create` a new user with the created `"changeOwnPasswordCustomDataRole"` role. For example, the following operation creates a user with both the built-in role `readWrite` (page 405) and the user-created `"changeOwnPasswordCustomDataRole"`.

```
use test
db.createUser(
   {
     user:"user123",
     pwd:"12345678",
     roles:[ "readWrite", { role:"changeOwnPasswordCustomDataRole", db:"admin" } ]
   }
)
```

To grant an existing user the new role, use `db.grantRolesToUser()`.

Procedure

Step 1: Connect with the appropriate privileges. Connect to the `mongod` or `mongos` as a user with appropriate privileges.

For example, the following operation connects to MongoDB as `user123` created in the *Prerequisites* (page 395) section.

```
mongo --port 27017 -u user123 -p 12345678 --authenticationDatabase test
```

To check that you have the privileges specified in the *Prerequisites* (page 395) section as well as to see user information, use the `usersInfo` command with the `showPrivileges` option.

Step 2: Change your password and custom data. Use the `db.updateUser()` method to update the password and custom data.

For example, the following operation changes thw user's password to `KNlZmiaNUp0B` and custom data to `{ title: "Senior Manager" }`:

```
use test
db.updateUser(
   "user123",
   {
      pwd: "KNlZmiaNUp0B",
      customData: { title: "Senior Manager" }
```

```
    }
)
```

6.3.5 Auditing Tutorials

The following tutorials provide instructions on how to enable auditing for system events and specify which events to audit.

Configure System Events Auditing **(page 397)** Enable and configure MongoDB Enterprise system event auditing feature.

Configure Audit Filters **(page 399)** Specify which events to audit.

Configure System Events Auditing

On this page

New in version 2.6.

MongoDB Enterprise[69] supports *auditing* (page 325) of various operations. A complete auditing solution must involve **all** `mongod` server and `mongos` router processes.

The audit facility can write audit events to the console, the *syslog* (option is unavailable on Windows), a JSON file, or a BSON file. For details on the audited operations and the audit log messages, see *System Event Audit Messages* (page 424).

Enable and Configure Audit Output

Use the `--auditDestination` option to enable auditing and specify where to output the audit events.

> **Warning:** For sharded clusters, if you enable auditing on `mongos` instances, you must enable auditing on all `mongod` instances in the cluster, i.e. shards and config servers.

Output to Syslog To enable auditing and print audit events to the syslog (option is unavailable on Windows) in JSON format, specify `syslog` for the `--auditDestination` setting. For example:

```
mongod --dbpath data/db --auditDestination syslog
```

> **Warning:** The syslog message limit can result in the truncation of the audit messages. The auditing system will neither detect the truncation nor error upon its occurrence.

You may also specify these options in the `configuration file`:

```
storage:
   dbPath: data/db
auditLog:
   destination: syslog
```

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

Output to Console To enable auditing and print the audit events to standard output (i.e. `stdout`), specify `console` for the *`--auditDestination`* setting. For example:

```
mongod --dbpath data/db --auditDestination console
```

You may also specify these options in the `configuration file`:

```
storage:
   dbPath: data/db
auditLog:
   destination: console
```

Output to JSON File To enable auditing and print audit events to a file in JSON format, specify `file` for the *`--auditDestination`* setting, `JSON` for the *`--auditFormat`* setting, and the output filename for the *`--auditPath`*. The *`--auditPath`* option accepts either full path name or relative path name. For example, the following enables auditing and records audit events to a file with the relative path name of `data/db/auditLog.json`:

```
mongod --dbpath data/db --auditDestination file --auditFormat JSON --auditPath data/db/auditLog.json
```

The audit file rotates at the same time as the server log file.

You may also specify these options in the `configuration file`:

```
storage:
   dbPath: data/db
auditLog:
   destination: file
   format: JSON
   path: data/db/auditLog.json
```

Note: Printing audit events to a file in JSON format degrades server performance more than printing to a file in BSON format.

Output to BSON File To enable auditing and print audit events to a file in BSON binary format, specify `file` for the *`--auditDestination`* setting, `BSON` for the *`--auditFormat`* setting, and the output filename for the *`--auditPath`*. The *`--auditPath`* option accepts either full path name or relative path name. For example, the following enables auditing and records audit events to a BSON file with the relative path name of `data/db/auditLog.bson`:

```
mongod --dbpath data/db --auditDestination file --auditFormat BSON --auditPath data/db/auditLog.bson
```

The audit file rotates at the same time as the server log file.

You may also specify these options in the `configuration file`:

```
storage:
   dbPath: data/db
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
```

To view the contents of the file, pass the file to the MongoDB utility `bsondump`. For example, the following converts the audit log into a human-readable form and output to the terminal:

```
bsondump data/db/auditLog.bson
```

See also:

Configure Audit Filters (page 399), *Auditing* (page 325), *System Event Audit Messages* (page 424)

Configure Audit Filters

On this page

MongoDB Enterprise[70] supports *auditing* (page 325) of various operations. When *enabled* (page 397), the audit facility, by default, records all auditable operations as detailed in *Audit Event Actions, Details, and Results* (page 425). To specify which events to record, the audit feature includes the *--auditFilter* option.

`--auditFilter` Option

The *--auditFilter* option takes a string representation of a query document of the form:

```
{ <field1>: <expression1>, ... }
```

- The `<field>` can be *any field in the audit message* (page 424), including fields returned in the *param* (page 425) document.
- The `<expression>` is a *query condition expression*.

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

Examples

Filter for Multiple Operation Types The following example audits only the `createCollection` (page 419) and `dropCollection` (page 420) actions by using the filter:

```
{ atype: { $in: [ "createCollection", "dropCollection" ] } }
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auditDestination file --auditFilter '{ atype: { $in: [ "createCollection",
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ atype: { $in: [ "createCollection", "dropCollection" ] } }'
```

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

Filter on Authentication Operations on a Single Database The `<field>` can include *any field in the audit message* (page 424). For authentication operations (i.e. `atype: "authenticate"`), the audit messages include a `db` field in the `param` document.

The following example audits only the `authenticate` operations that occur against the `test` database by using the filter:

```
{ atype: "authenticate", "param.db": "test" }
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ atype: "authenticate", "param
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
security:
   authorization: enabled
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ atype: "authenticate", "param.db": "test" }'
```

To filter on all `authenticate` operations across databases, use the filter `{ atype: "authenticate" }`.

Filter on Collection Creation and Drop Operations for a Single Database The `<field>` can include *any field in the audit message* (page 424). For collection creation and drop operations (i.e. `atype: "createCollection"` and `atype: "dropCollection"`), the audit messages include a namespace `ns` field in the `param` document.

The following example audits only the `createCollection` and `dropCollection` operations that occur against the `test` database by using the filter:

Note: The regular expression requires two backslashes (`\\`) to escape the dot (`.`).

```
{ atype: { $in: [ "createCollection", "dropCollection" ] }, "param.ns": /^test\\./ } }
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ atype: { $in: [ "createColle
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
security:
   authorization: enabled
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ atype: { $in: [ "createCollection", "dropCollection" ] }, "param.ns": /^test\\./ } }'
```

Filter by Authorization Role The following example audits operations by users with `readWrite` (page 405) role on the `test` database, including users with roles that inherit from `readWrite` (page 405), by using the filter:

```
{ roles: { role: "readWrite", db: "test" } }
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ roles: { role: "readWrite", c
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
security:
   authorization: enabled
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ roles: { role: "readWrite", db: "test" } }'
```

Filter on Read and Write Operations To capture `read` and `write` operations in the audit, you must also enable the audit system to log authorization successes using the `auditAuthorizationSuccess` parameter. [71]

Note: Enabling `auditAuthorizationSuccess` degrades performance more than logging only the authorization failures.

The following example audits the `find()`, `insert()`, `remove()`, `update()`, `save()`, and `findAndModify()` operations by using the filter:

```
{ atype: "authCheck", "param.command": { $in: [ "find", "insert", "delete", "update", "findandmodify"
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auth --setParameter auditAuthorizationSuccess=true --auditDestination file
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
security:
   authorization: enabled
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ atype: "authCheck", "param.command": { $in: [ "find", "insert", "delete", "update", "fi
setParameter: { auditAuthorizationSuccess: true }
```

Filter on Read and Write Operations for a Collection To capture `read` and `write` operations in the audit, you must also enable the audit system to log authorization successes using the `auditAuthorizationSuccess` parameter. [1]

Note: Enabling `auditAuthorizationSuccess` degrades performance more than logging only the authorization failures.

---

[71] You can enable `auditAuthorizationSuccess` parameter without enabling `--auth`; however, all operations will return success for authorization checks.

The following example audits the `find()`, `insert()`, `remove()`, `update()`, `save()`, and `findAndModify()` operations for the collection `orders` in the database `test` by using the filter:

```
{ atype: "authCheck", "param.ns": "test.orders", "param.command": { $in: [ "find", "insert", "delete"
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```
mongod --dbpath data/db --auth --setParameter auditAuthorizationSuccess=true --auditDestination file
```

To specify the audit filter in a `configuration file`, you must use the YAML format of the configuration file.

```
storage:
   dbPath: data/db
security:
   authorization: enabled
auditLog:
   destination: file
   format: BSON
   path: data/db/auditLog.bson
   filter: '{ atype: "authCheck", "param.ns": "test.orders", "param.command": { $in: [ "find", "inse
setParameter: { auditAuthorizationSuccess: true }
```

See also:

Configure System Events Auditing (page 397), *Auditing* (page 325), *System Event Audit Messages* (page 424)

6.3.6 Create a Vulnerability Report

On this page

If you believe you have discovered a vulnerability in MongoDB or have experienced a security incident related to MongoDB, please report the issue to aid in its resolution.

To report an issue, we strongly suggest filing a ticket in the SECURITY[72] project in JIRA. MongoDB, Inc responds to vulnerability notifications within 48 hours.

Create the Report in JIRA

Submit a ticket in the Security[73] project at: <http://jira.mongodb.org/browse>. The ticket number will become the reference identification for the issue for its lifetime. You can use this identifier for tracking purposes.

Information to Provide

All vulnerability reports should contain as much information as possible so MongoDB's developers can move quickly to resolve the issue. In particular, please include the following:

[72] https://jira.mongodb.org/browse/SECURITY
[73] https://jira.mongodb.org/browse/SECURITY

- The name of the product.
- *Common Vulnerability* information, if applicable, including:
- CVSS (Common Vulnerability Scoring System) Score.
- CVE (Common Vulnerability and Exposures) Identifier.
- Contact information, including an email address and/or phone number, if applicable.

Send the Report via Email

While JIRA is the preferred reporting method, you may also report vulnerabilities via email to security@mongodb.com[74].

You may encrypt email using MongoDB's public key at https://docs.mongodb.org/10gen-security-gpg-key.asc.

MongoDB, Inc. responds to vulnerability reports sent via email with a response email that contains a reference number for a JIRA ticket posted to the SECURITY[75] project.

Evaluation of a Vulnerability Report

MongoDB, Inc. validates all submitted vulnerabilities and uses Jira to track all communications regarding a vulnerability, including requests for clarification or additional information. If needed, MongoDB representatives set up a conference call to exchange information regarding the vulnerability.

Disclosure

MongoDB, Inc. requests that you do *not* publicly disclose any information regarding the vulnerability or exploit the issue until it has had the opportunity to analyze the vulnerability, to respond to the notification, and to notify key users, customers, and partners.

The amount of time required to validate a reported vulnerability depends on the complexity and severity of the issue. MongoDB, Inc. takes all required vulnerabilities very seriously and will always ensure that there is a clear and open channel of communication with the reporter.

After validating an issue, MongoDB, Inc. coordinates public disclosure of the issue with the reporter in a mutually agreed timeframe and format. If required or requested, the reporter of a vulnerability will receive credit in the published security bulletin.

6.4 Security Reference

On this page

[74] security@mongodb.com
[75] https://jira.mongodb.org/browse/SECURITY

6.4.1 Security Methods in the mongo Shell

Name	Description
db.auth()	Authenticates a user to a database.

User Management Methods

Name	Description
db.createUser()	Creates a new user.
db.addUser()	Deprecated. Adds a user to a database, and allows administrators to configure the user's privileges.
db.updateUser()	Updates user data.
db.changeUserPassword()	Changes an existing user's password.
db.removeUser()	Deprecated. Removes a user from a database.
db.dropAllUsers()	Deletes all users associated with a database.
db.dropUser()	Removes a single user.
db.grantRolesToUser()	Grants a role and its privileges to a user.
db.revokeRolesFromUser()	Removes a role from a user.
db.getUser()	Returns information about the specified user.
db.getUsers()	Returns information about all users associated with a database.

Role Management Methods

Name	Description
db.createRole()	Creates a role and specifies its privileges.
db.updateRole()	Updates a user-defined role.
db.dropRole()	Deletes a user-defined role.
db.dropAllRoles()	Deletes all user-defined roles associated with a database.
db.grantPrivilegesToRole()	Assigns privileges to a user-defined role.
db.revokePrivilegesFromRole()	Removes the specified privileges from a user-defined role.
db.grantRolesToRole()	Specifies roles from which a user-defined role inherits privileges.
db.revokeRolesFromRole()	Removes inherited roles from a role.
db.getRole()	Returns information for the specified role.
db.getRoles()	Returns information for all the user-defined roles in a database.

6.4.2 Security Reference Documentation

Built-In Roles **(page 405)** Reference on MongoDB provided roles and corresponding access.

system.roles Collection **(page 412)** Describes the content of the collection that stores user-defined roles.

system.users Collection **(page 415)** Describes the content of the collection that stores users' credentials and role assignments.

Resource Document **(page 417)** Describes the resource document for roles.

Privilege Actions **(page 418)** List of the actions available for privileges.

Default MongoDB Port **(page 424)** List of default ports used by MongoDB.

System Event Audit Messages **(page 424)** Reference on system event audit messages.

Built-In Roles

On this page

MongoDB grants access to data and commands through *role-based authorization* (page 320) and provides built-in roles that provide the different levels of access commonly needed in a database system. You can additionally create *user-defined roles* (page 321).

A role grants privileges to perform sets of *actions* (page 418) on defined *resources* (page 417). A given role applies to the database on which it is defined and can grant access down to a collection level of granularity.

Each of MongoDB's built-in roles defines access at the database level for all *non*-system collections in the role's database and at the collection level for all *system collections* (page 304).

MongoDB provides the built-in *database user* (page 405) and *database administration* (page 406) roles on *every* database. MongoDB provides all other built-in roles only on the `admin` database.

This section describes the privileges for each built-in role. You can also view the privileges for a built-in role at any time by issuing the `rolesInfo` command with the `showPrivileges` and `showBuiltinRoles` fields both set to `true`.

Database User Roles

Every database includes the following client roles:

`read`

Provides the ability to read data on all *non*-system collections and on the following system collections: `system.indexes` (page 304), `system.js` (page 304), and `system.namespaces` (page 304) collections. The role provides read access by granting the following *actions* (page 418):

- `collStats` (page 423)
- `dbHash` (page 423)
- `dbStats` (page 423)
- `find` (page 419)
- `killCursors` (page 420)

`readWrite`

Provides all the privileges of the `read` (page 405) role plus ability to modify data on all *non*-system collections and the `system.js` (page 304) collection. The role provides the following actions on those collections:

- `collStats` (page 423)
- `convertToCapped` (page 422)
- `createCollection` (page 419)
- `dbHash` (page 423)

- dbStats (page 423)
- dropCollection (page 420)
- createIndex (page 419)
- dropIndex (page 422)
- emptycapped (page 420)
- find (page 419)
- insert (page 419)
- killCursors (page 420)
- remove (page 419)
- renameCollectionSameDB (page 422)
- update (page 419)

Database Administration Roles

Every database includes the following database administration roles:

dbAdmin

Provides the following *actions* (page 418) on the database's system.indexes (page 304), system.namespaces (page 304), and system.profile (page 304) collections:

- collStats (page 423)
- dbHash (page 423)
- dbStats (page 423)
- find (page 419)
- killCursors (page 420)
- dropCollection (page 420) and createCollection (page 419) on system.profile (page 304) *only*

Changed in version 2.6.4: dbAdmin (page 406) added the createCollection (page 419) for the system.profile (page 304) collection. Previous versions only had the dropCollection (page 420) on the system.profile (page 304) collection.

Provides the following actions on all *non*-system collections. This role *does not* include full read access on non-system collections:

- collMod (page 422)
- collStats (page 423)
- compact (page 422)
- convertToCapped (page 422)
- createCollection (page 419)
- createIndex (page 419)
- dbStats (page 423)
- dropCollection (page 420)
- dropDatabase (page 422)

- dropIndex (page 422)
- enableProfiler (page 420)
- indexStats (page 423)
- reIndex (page 422)
- renameCollectionSameDB (page 422)
- repairDatabase (page 423)
- storageDetails (page 421)
- validate (page 423)

dbOwner

The database owner can perform any administrative action on the database. This role combines the privileges granted by the readWrite (page 405), dbAdmin (page 406) and userAdmin (page 407) roles.

userAdmin

Provides the ability to create and modify roles and users on the current database. This role also indirectly provides *superuser* (page 412) access to either the database or, if scoped to the admin database, the cluster. The userAdmin (page 407) role allows users to grant any user any privilege, including themselves.

The userAdmin (page 407) role explicitly provides the following actions:

- changeCustomData (page 419)
- changePassword (page 419)
- createRole (page 420)
- createUser (page 420)
- dropRole (page 420)
- dropUser (page 420)
- grantRole (page 420)
- revokeRole (page 420)
- viewRole (page 420)
- viewUser (page 420)

Cluster Administration Roles

The admin database includes the following roles for administering the whole system rather than just a single database. These roles include but are not limited to *replica set* and *sharded cluster* administrative functions.

clusterAdmin

Provides the greatest cluster-management access. This role combines the privileges granted by the clusterManager (page 407), clusterMonitor (page 408), and hostManager (page 409) roles. Additionally, the role provides the dropDatabase (page 422) action.

clusterManager

Provides management and monitoring actions on the cluster. A user with this role can access the config and local databases, which are used in sharding and replication, respectively.

Provides the following actions on the cluster as a whole:

- addShard (page 421)

- applicationMessage (page 422)
- cleanupOrphaned (page 420)
- flushRouterConfig (page 421)
- listShards (page 421)
- removeShard (page 422)
- replSetConfigure (page 421)
- replSetGetStatus (page 421)
- replSetStateChange (page 421)
- resync (page 421)

Provides the following actions on *all* databases in the cluster:

- enableSharding (page 421)
- moveChunk (page 421)
- splitChunk (page 422)
- splitVector (page 422)

On the config database, provides the following actions on the settings (page 758) collection:

- insert (page 419)
- remove (page 419)
- update (page 419)

On the config database, provides the following actions on all configuration collections and on the system.indexes (page 304), system.js (page 304), and system.namespaces (page 304) collections:

- collStats (page 423)
- dbHash (page 423)
- dbStats (page 423)
- find (page 419)
- killCursors (page 420)

On the local database, provides the following actions on the replset (page 666) collection:

- collStats (page 423)
- dbHash (page 423)
- dbStats (page 423)
- find (page 419)
- killCursors (page 420)

clusterMonitor

Provides read-only access to monitoring tools, such as the MongoDB Cloud Manager[76] monitoring agent.

Provides the following actions on the cluster as a whole:

- connPoolStats (page 423)

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

- cursorInfo (page 423)
- getCmdLineOpts (page 423)
- getLog (page 423)
- getParameter (page 422)
- getShardMap (page 421)
- hostInfo (page 422)
- inprog (page 420)
- listDatabases (page 423)
- listShards (page 421)
- netstat (page 423)
- replSetGetStatus (page 421)
- serverStatus (page 423)
- shardingState (page 422)
- top (page 423)

Provides the following actions on *all* databases in the cluster:

- collStats (page 423)
- dbStats (page 423)
- getShardVersion (page 421)

Provides the find (page 419) action on all system.profile (page 304) collections in the cluster.

Provides the following actions on the config database's configuration collections and system.indexes (page 304), system.js (page 304), and system.namespaces (page 304) collections:

- collStats (page 423)
- dbHash (page 423)
- dbStats (page 423)
- find (page 419)
- killCursors (page 420)

hostManager

Provides the ability to monitor and manage servers.

Provides the following actions on the cluster as a whole:

- applicationMessage (page 422)
- closeAllDatabases (page 422)
- connPoolSync (page 422)
- cpuProfiler (page 420)
- diagLogging (page 423)
- flushRouterConfig (page 421)
- fsync (page 422)
- invalidateUserCache (page 420)

- killop (page 421)
- logRotate (page 422)
- resync (page 421)
- setParameter (page 423)
- shutdown (page 423)
- touch (page 423)
- unlock (page 420)

Provides the following actions on *all* databases in the cluster:

- killCursors (page 420)
- repairDatabase (page 423)

Backup and Restoration Roles

The admin database includes the following roles for backing up and restoring data:

backup

Provides minimal privileges needed for backing up data. This role provides sufficient privileges to use the MongoDB Cloud Manager[77] backup agent, or to use mongodump to back up an entire mongod instance.

Provides the following *actions* (page 418) on the mms.backup collection in the admin database:

- insert (page 419)
- update (page 419)

Provides the listDatabases (page 423) action on the cluster as a whole.

Provides the find (page 419) action on the following:

- all *non*-system collections in the cluster
- all the following system collections in the cluster: system.indexes (page 304), system.namespaces (page 304), and system.js (page 304)
- the admin.system.users (page 304) and admin.system.roles (page 304) collections
- legacy system.users collections from versions of MongoDB prior to 2.6

restore

Provides privileges needed to restore data from backups. This role is sufficient when restoring data with mongorestore without the *--oplogReplay* option. If running mongorestore with *--oplogReplay*, however, the restore (page 410) role is insufficient to replay the oplog. To replay the oplog, create a *user-defined role* that has anyAction (page 424) on *anyResource* (page 418) and grant only to users who must run mongorestore with *--oplogReplay*.

Provides the following actions on all *non*-system collections and system.js (page 304) collections in the cluster; on the admin.system.users (page 304) and admin.system.roles (page 304) collections in the admin database; and on legacy system.users collections from versions of MongoDB prior to 2.6:

- collMod (page 422)
- createCollection (page 419)
- createIndex (page 419)

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

- dropCollection (page 420)
- insert (page 419)

Provides the following *additional* actions on admin.system.users (page 304) and legacy system.users collections:

- find (page 419)
- remove (page 419)
- update (page 419)

Provides the find (page 419) action on all the system.namespaces (page 304) collections in the cluster.

Although, restore (page 410) includes the ability to modify the documents in the admin.system.users (page 304) collection using normal modification operations, *only* modify these data using the *user management methods*.

All-Database Roles

The admin database provides the following roles that apply to all databases in a mongod instance and are roughly equivalent to their single-database equivalents:

readAnyDatabase
Provides the same read-only permissions as read (page 405), except it applies to *all* databases in the cluster. The role also provides the listDatabases (page 423) action on the cluster as a whole.

readWriteAnyDatabase
Provides the same read and write permissions as readWrite (page 405), except it applies to *all* databases in the cluster. The role also provides the listDatabases (page 423) action on the cluster as a whole.

userAdminAnyDatabase
Provides the same access to user administration operations as userAdmin (page 407), except it applies to *all* databases in the cluster. The role also provides the following actions on the cluster as a whole:

- authSchemaUpgrade (page 420)
- invalidateUserCache (page 420)
- listDatabases (page 423)

The role also provides the following actions on the admin.system.users (page 304) and admin.system.roles (page 304) collections on the admin database, and on legacy system.users collections from versions of MongoDB prior to 2.6:

- collStats (page 423)
- dbHash (page 423)
- dbStats (page 423)
- find (page 419)
- killCursors (page 420)
- planCacheRead (page 421)

Changed in version 2.6.4: userAdminAnyDatabase (page 411) added the following permissions on the admin.system.users (page 304) and admin.system.roles (page 304) collections:

- createIndex (page 419)
- dropIndex (page 422)

The userAdminAnyDatabase (page 411) role does not restrict the permissions that a user can grant. As a result, userAdminAnyDatabase (page 411) users can grant themselves privileges in excess of their current privileges and even can grant themselves *all privileges*, even though the role does not explicitly authorize privileges beyond user administration. This role is effectively a MongoDB system *superuser* (page 412).

dbAdminAnyDatabase
Provides the same access to database administration operations as dbAdmin (page 406), except it applies to *all* databases in the cluster. The role also provides the listDatabases (page 423) action on the cluster as a whole.

Superuser Roles

Several roles provide either indirect or direct system-wide superuser access.

The following roles provide the ability to assign any user any privilege on any database, which means that users with one of these roles can assign *themselves* any privilege on any database:

- dbOwner (page 407) role, when scoped to the admin database
- userAdmin (page 407) role, when scoped to the admin database
- userAdminAnyDatabase (page 411) role

The following role provides full privileges on all resources:

root
Provides access to the operations and all the resources of the readWriteAnyDatabase (page 411), dbAdminAnyDatabase (page 412), userAdminAnyDatabase (page 411) and clusterAdmin (page 407) roles *combined*.

root (page 412) does **not** include any access to collections that begin with the system. prefix.

For example, without the ability to insert data directly into the:data:*system.users* <*admin.system.users*> and system.roles (page 304) collections in the admin database. root (page 412) is not suitable for writing or restoring data that have these collections (e.g. with mongorestore.) To perform these kinds of restore operations, provision users with the restore (page 410) role.

Internal Role

__system
MongoDB assigns this role to user objects that represent cluster members, such as replica set members and mongos instances. The role entitles its holder to take any action against any object in the database.

Do not assign this role to user objects representing applications or human administrators, other than in exceptional circumstances.

If you need access to all actions on all resources, for example to run the eval or applyOps commands, do not assign this role. Instead, *create a user-defined role* (page 386) that grants anyAction (page 424) on *anyResource* (page 418) and ensure that only the users who needs access to these operations has this access.

system.roles Collection

New in version 2.6.

On this page

The system.roles collection in the admin database stores the user-defined roles. To create and manage these user-defined roles, MongoDB provides *role management commands*.

system.roles Schema

The documents in the system.roles collection have the following schema:

```
{
  _id: <system-defined id>,
  role: "<role name>",
  db: "<database>",
  privileges:
      [
          {
              resource: { <resource> },
              actions: [ "<action>", ... ]
          },
          ...
      ],
  roles:
      [
          { role: "<role name>", db: "<database>" },
          ...
      ]
}
```

A system.roles document has the following fields:

admin.system.roles.**role**
: The role (page 413) field is a string that specifies the name of the role.

admin.system.roles.**db**
: The db (page 413) field is a string that specifies the database to which the role belongs. MongoDB uniquely identifies each role by the pairing of its name (i.e. role (page 413)) and its database.

admin.system.roles.**privileges**
: The privileges (page 413) array contains the privilege documents that define the *privileges* (page 320) for the role.

  A privilege document has the following syntax:

  ```
  {
    resource: { <resource> },
    actions: [ "<action>", ... ]
  }
  ```

  Each privilege document has the following fields:

  admin.system.roles.privileges[n].**resource**
  : A document that specifies the resources upon which the privilege actions (page 414) apply. The document has one of the following form:

```
{ db: <database>, collection: <collection> }
```

or

```
{ cluster : true }
```

See *Resource Document* (page 417) for more details.

`admin.system.roles.privileges[n].actions`
An array of actions permitted on the resource. For a list of actions, see *Privilege Actions* (page 418).

`admin.system.roles.roles`
The `roles` (page 414) array contains role documents that specify the roles from which this role *inherits* (page 321) privileges.

A role document has the following syntax:

```
{ role: "<role name>", db: "<database>" }
```

A role document has the following fields:

`admin.system.roles.roles[n].role`
The name of the role. A role can be a *built-in role* (page 405) provided by MongoDB or a *user-defined role* (page 321).

`admin.system.roles.roles[n].db`
The name of the database where the role is defined.

Examples

Consider the following sample documents found in `system.roles` collection of the `admin` database.

A User-Defined Role Specifies Privileges The following is a sample document for a user-defined role `appUser` defined for the `myApp` database:

```
{
  _id: "myApp.appUser",
  role: "appUser",
  db: "myApp",
  privileges: [
       { resource: { db: "myApp" , collection: "" },
         actions: [ "find", "createCollection", "dbStats", "collStats" ] },
       { resource: { db: "myApp", collection: "logs" },
         actions: [ "insert" ] },
       { resource: { db: "myApp", collection: "data" },
         actions: [ "insert", "update", "remove", "compact" ] },
       { resource: { db: "myApp", collection: "system.indexes" },
         actions: [ "find" ] },
       { resource: { db: "myApp", collection: "system.namespaces" },
         actions: [ "find" ] },
  ],
  roles: []
}
```

The `privileges` array lists the five privileges that the `appUser` role specifies:

- The first privilege permits its actions (`"find"`, `"createCollection"`, `"dbStats"`, `"collStats"`) on all the collections in the `myApp` database *excluding* its system collections. See *Specify a Database as Resource* (page 417).

- The next two privileges permits *additional* actions on specific collections, `logs` and `data`, in the `myApp` database. See *Specify a Collection of a Database as Resource* (page 417).
- The last two privileges permits actions on two *system collections* (page 304) in the `myApp` database. While the first privilege gives database-wide permission for the `find` action, the action does not apply to `myApp`'s system collections. To give access to a system collection, a privilege must explicitly specify the collection. See *Resource Document* (page 417).

As indicated by the empty `roles` array, `appUser` inherits no additional privileges from other roles.

User-Defined Role Inherits from Other Roles The following is a sample document for a user-defined role `appAdmin` defined for the `myApp` database: The document shows that the `appAdmin` role specifies privileges as well as inherits privileges from other roles:

```
{
  _id: "myApp.appAdmin",
  role: "appAdmin",
  db: "myApp",
  privileges: [
                {
                  resource: { db: "myApp", collection: "" },
                  actions: [ "insert", "dbStats", "collStats", "compact", "repairDatabase" ]
                }
              ],
  roles: [
           { role: "appUser", db: "myApp" }
         ]
}
```

The `privileges` array lists the privileges that the `appAdmin` role specifies. This role has a single privilege that permits its actions (`"insert"`, `"dbStats"`, `"collStats"`, `"compact"`, `"repairDatabase"`) on all the collections in the `myApp` database *excluding* its system collections. See *Specify a Database as Resource* (page 417).

The `roles` array lists the roles, identified by the role names and databases, from which the role `appAdmin` inherits privileges.

`system.users` Collection

Changed in version 2.6.

On this page

The `system.users` collection in the `admin` database stores user *authentication* (page 316) and *authorization* (page 320) information. To manage data in this collection, MongoDB provides *user management commands*.

`system.users` Schema

The documents in the `system.users` collection have the following schema:

```
{
  _id: <system defined id>,
  user: "<name>",
```

```
  db: "<database>",
  credentials: { <authentication credentials> },
  roles: [
           { role: "<role name>", db: "<database>" },
           ...
         ],
  customData: <custom information>
 }
```

Each `system.users` document has the following fields:

`admin.system.users.`**`user`**
: The `user` (page 416) field is a string that identifies the user. A user exists in the context of a single logical database but can have access to other databases through roles specified in the `roles` (page 416) array.

`admin.system.users.`**`db`**
: The `db` (page 416) field specifies the database associated with the user. The user's privileges are not necessarily limited to this database. The user can have privileges in additional databases through the `roles` (page 416) array.

`admin.system.users.`**`credentials`**
: The `credentials` (page 416) field contains the user's authentication information. For users with externally stored authentication credentials, such as users that use *Kerberos* (page 369) or x.509 certificates for authentication, the `system.users` document for that user does not contain the `credentials` (page 416) field.

`admin.system.users.`**`roles`**
: The `roles` (page 416) array contains role documents that specify the roles granted to the user. The array contains both *built-in roles* (page 405) and *user-defined role* (page 321).

  A role document has the following syntax:

  ```
  { role: "<role name>", db: "<database>" }
  ```

  A role document has the following fields:

  `admin.system.users.roles[n].`**`role`**
  : The name of a role. A role can be a *built-in role* (page 405) provided by MongoDB or a *custom user-defined role* (page 321).

  `admin.system.users.roles[n].`**`db`**
  : The name of the database where role is defined.

  When specifying a role using the *role management* or *user management* commands, you can specify the role name alone (e.g. `"readWrite"`) if the role that exists on the database on which the command is run.

`admin.system.users.`**`customData`**
: The `customData` (page 416) field contains optional custom information about the user.

Example

Consider the following document in the `system.users` collection:

```
{
  _id: "home.Kari",
  user: "Kari",
  db: "home",
  credentials: { "MONGODB-CR" :"<hashed password>" },
  roles : [
            { role: "read", db: "home" },
```

```
            { role: "readWrite", db: "test" },
            { role: "appUser", db: "myApp" }
          ],
  customData: { zipCode: "64157" }
}
```

The document shows that a user `Kari` is associated with the `home` database. `Kari` has the `read` (page 405) role in the `home` database, the `readWrite` (page 405) role in the `test` database, and the `appUser` role in the `myApp` database.

Resource Document

On this page

The resource document specifies the resources upon which a privilege permits `actions`.

Database and/or Collection Resource

To specify databases and/or collections, use the following syntax:

```
{ db: <database>, collection: <collection> }
```

Specify a Collection of a Database as Resource If the resource document species both the `db` and `collection` fields as non-empty strings, the resource is the specified collection in the specified database. For example, the following document specifies a resource of the `inventory` collection in the `products` database:

```
{ db: "products", collection: "inventory" }
```

For a user-defined role scoped for a non-`admin` database, the resource specification for its privileges must specify the same database as the role. User-defined roles scoped for the `admin` database can specify other databases.

Specify a Database as Resource If only the `collection` field is an empty string (`""`), the resource is the specified database, excluding the *system collections* (page 304). For example, the following resource document specifies the resource of the `test` database, excluding the system collections:

```
{ db: "test", collection: "" }
```

For a user-defined role scoped for a non-`admin` database, the resource specification for its privileges must specify the same database as the role. User-defined roles scoped for the `admin` database can specify other databases.

Note: When you specify a database as the resource, the system collections are excluded, unless you name them explicitly, as in the following:

```
{ db: "test", collection: "system.namespaces" }
```

System collections include but are not limited to the following:

- `<database>.system.profile` (page 304)

- <database>.system.namespaces (page 304)
- <database>.system.indexes (page 304)
- <database>.system.js (page 304)
- local.system.replset (page 666)
- *system.users Collection* (page 415) in the admin database
- *system.roles Collection* (page 412) in the admin database

Specify Collections Across Databases as Resource If only the db field is an empty string (""), the resource is all collections with the specified name across all databases. For example, the following document specifies the resource of all the accounts collections across all the databases:

```
{ db: "", collection: "accounts" }
```

For user-defined roles, only roles scoped for the admin database can have this resource specification for their privileges.

Specify All Non-System Collections in All Databases If both the db and collection fields are empty strings (""), the resource is all collections, excluding the *system collections* (page 304), in all the databases:

```
{ db: "", collection: "" }
```

For user-defined roles, only roles scoped for the admin database can have this resource specification for their privileges.

Cluster Resource

To specify the cluster as the resource, use the following syntax:

```
{ cluster : true }
```

Use the cluster resource for actions that affect the state of the system rather than act on specific set of databases or collections. Examples of such actions are shutdown, replSetReconfig, and addShard. For example, the following document grants the action shutdown on the cluster.

```
{ resource: { cluster : true }, actions: [ "shutdown" ] }
```

For user-defined roles, only roles scoped for the admin database can have this resource specification for their privileges.

anyResource

The internal resource anyResource gives access to every resource in the system and is intended for internal use. **Do not** use this resource, other than in exceptional circumstances. The syntax for this resource is { anyResource: true }.

Privilege Actions

New in version 2.6.

On this page

Privilege actions define the operations a user can perform on a *resource* (page 417). A MongoDB *privilege* (page 320) comprises a *resource* (page 417) and the permitted actions. This page lists available actions grouped by common purpose.

MongoDB provides built-in roles with pre-defined pairings of resources and permitted actions. For lists of the actions granted, see *Built-In Roles* (page 405). To define custom roles, see *Create a Role* (page 386).

Query and Write Actions

find
User can perform the `db.collection.find()` method. Apply this action to database or collection resources.

insert
User can perform the `insert` command. Apply this action to database or collection resources.

remove
User can perform the `db.collection.remove()` method. Apply this action to database or collection resources.

update
User can perform the `update` command. Apply this action to database or collection resources.

Database Management Actions

changeCustomData
User can change the custom information of any user in the given database. Apply this action to database resources.

changeOwnCustomData
Users can change their own custom information. Apply this action to database resources. See also *Change Your Password and Custom Data* (page 395).

changeOwnPassword
Users can change their own passwords. Apply this action to database resources. See also *Change Your Password and Custom Data* (page 395).

changePassword
User can change the password of any user in the given database. Apply this action to database resources.

createCollection
User can perform the `db.createCollection()` method. Apply this action to database or collection resources.

createIndex
Provides access to the `db.collection.createIndex()` method and the `createIndexes` command. Apply this action to database or collection resources.

createRole
User can create new roles in the given database. Apply this action to database resources.

createUser
User can create new users in the given database. Apply this action to database resources.

dropCollection
User can perform the `db.collection.drop()` method. Apply this action to database or collection resources.

dropRole
User can delete any role from the given database. Apply this action to database resources.

dropUser
User can remove any user from the given database. Apply this action to database resources.

emptycapped
User can perform the `emptycapped` command. Apply this action to database or collection resources.

enableProfiler
User can perform the `db.setProfilingLevel()` method. Apply this action to database resources.

grantRole
User can grant any role in the database to any user from any database in the system. Apply this action to database resources.

killCursors
User can kill cursors on the target collection.

revokeRole
User can remove any role from any user from any database in the system. Apply this action to database resources.

unlock
User can perform the `db.fsyncUnlock()` method. Apply this action to the `cluster` resource.

viewRole
User can view information about any role in the given database. Apply this action to database resources.

viewUser
User can view the information of any user in the given database. Apply this action to database resources.

Deployment Management Actions

authSchemaUpgrade
User can perform the `authSchemaUpgrade` command. Apply this action to the `cluster` resource.

cleanupOrphaned
User can perform the `cleanupOrphaned` command. Apply this action to the `cluster` resource.

cpuProfiler
User can enable and use the CPU profiler. Apply this action to the `cluster` resource.

inprog
User can use the `db.currentOp()` method to return pending and active operations. Apply this action to the `cluster` resource.

invalidateUserCache
Provides access to the `invalidateUserCache` command. Apply this action to the `cluster` resource.

killop
User can perform the `db.killOp()` method. Apply this action to the `cluster` resource.

planCacheRead
User can perform the `planCacheListPlans` and `planCacheListQueryShapes` commands and the `PlanCache.getPlansByQuery()` and `PlanCache.listQueryShapes()` methods. Apply this action to database or collection resources.

planCacheWrite
User can perform the `planCacheClear` command and the `PlanCache.clear()` and `PlanCache.clearPlansByQuery()` methods. Apply this action to database or collection resources.

storageDetails
User can perform the `storageDetails` command. Apply this action to database or collection resources.

Replication Actions

appendOplogNote
User can append notes to the oplog. Apply this action to the `cluster` resource.

replSetConfigure
User can configure a replica set. Apply this action to the `cluster` resource.

replSetGetStatus
User can perform the `replSetGetStatus` command. Apply this action to the `cluster` resource.

replSetHeartbeat
User can perform the `replSetHeartbeat` command. Apply this action to the `cluster` resource.

replSetStateChange
User can change the state of a replica set through the `replSetFreeze`, `replSetMaintenance`, `replSetStepDown`, and `replSetSyncFrom` commands. Apply this action to the `cluster` resource.

resync
User can perform the `resync` command. Apply this action to the `cluster` resource.

Sharding Actions

addShard
User can perform the `addShard` command. Apply this action to the `cluster` resource.

enableSharding
User can enable sharding on a database using the `enableSharding` command and can shard a collection using the `shardCollection` command. Apply this action to database or collection resources.

flushRouterConfig
User can perform the `flushRouterConfig` command. Apply this action to the `cluster` resource.

getShardMap
User can perform the `getShardMap` command. Apply this action to the `cluster` resource.

getShardVersion
User can perform the `getShardVersion` command. Apply this action to database resources.

listShards
User can perform the `listShards` command. Apply this action to the `cluster` resource.

moveChunk
User can perform the `moveChunk` command. In addition, user can perform the `movePrimary` command

provided that the privilege is applied to an appropriate database resource. Apply this action to database or collection resources.

removeShard
User can perform the `removeShard` command. Apply this action to the `cluster` resource.

shardingState
User can perform the `shardingState` command. Apply this action to the `cluster` resource.

splitChunk
User can perform the `splitChunk` command. Apply this action to database or collection resources.

splitVector
User can perform the `splitVector` command. Apply this action to database or collection resources.

Server Administration Actions

applicationMessage
User can perform the `logApplicationMessage` command. Apply this action to the `cluster` resource.

closeAllDatabases
User can perform the `closeAllDatabases` command. Apply this action to the `cluster` resource.

collMod
User can perform the `collMod` command. Apply this action to database or collection resources.

compact
User can perform the `compact` command. Apply this action to database or collection resources.

connPoolSync
User can perform the `connPoolSync` command. Apply this action to the `cluster` resource.

convertToCapped
User can perform the `convertToCapped` command. Apply this action to database or collection resources.

dropDatabase
User can perform the `dropDatabase` command. Apply this action to database resources.

dropIndex
User can perform the `dropIndexes` command. Apply this action to database or collection resources.

fsync
User can perform the `fsync` command. Apply this action to the `cluster` resource.

getParameter
User can perform the `getParameter` command. Apply this action to the `cluster` resource.

hostInfo
Provides information about the server the MongoDB instance runs on. Apply this action to the `cluster` resource.

logRotate
User can perform the `logRotate` command. Apply this action to the `cluster` resource.

reIndex
User can perform the `reIndex` command. Apply this action to database or collection resources.

renameCollectionSameDB
Allows the user to rename collections on the current database using the `renameCollection` command. Apply this action to database resources.

Additionally, the user must either *have* `find` (page 419) on the source collection or *not have* `find` (page 419) on the destination collection.

If a collection with the new name already exists, the user must also have the `dropCollection` (page 420) action on the destination collection.

repairDatabase
User can perform the `repairDatabase` command. Apply this action to database resources.

setParameter
User can perform the `setParameter` command. Apply this action to the `cluster` resource.

shutdown
User can perform the `shutdown` command. Apply this action to the `cluster` resource.

touch
User can perform the `touch` command. Apply this action to the `cluster` resource.

Diagnostic Actions

collStats
User can perform the `collStats` command. Apply this action to database or collection resources.

connPoolStats
User can perform the `connPoolStats` and `shardConnPoolStats` commands. Apply this action to the `cluster` resource.

cursorInfo
User can perform the `cursorInfo` command. Apply this action to the `cluster` resource.

dbHash
User can perform the `dbHash` command. Apply this action to database or collection resources.

dbStats
User can perform the `dbStats` command. Apply this action to database resources.

diagLogging
User can perform the `diagLogging` command. Apply this action to the `cluster` resource.

getCmdLineOpts
User can perform the `getCmdLineOpts` command. Apply this action to the `cluster` resource.

getLog
User can perform the `getLog` command. Apply this action to the `cluster` resource.

indexStats
User can perform the `indexStats` command. Apply this action to database or collection resources.

listDatabases
User can perform the `listDatabases` command. Apply this action to the `cluster` resource.

netstat
User can perform the `netstat` command. Apply this action to the `cluster` resource.

serverStatus
User can perform the `serverStatus` command. Apply this action to the `cluster` resource.

validate
User can perform the `validate` command. Apply this action to database or collection resources.

top
User can perform the `top` command. Apply this action to the `cluster` resource.

Internal Actions

anyAction
Allows any action on a resource. **Do not** assign this action except for exceptional circumstances.

internal
Allows internal actions. **Do not** assign this action except for exceptional circumstances.

Default MongoDB Port

The following table lists the default TCP ports used by MongoDB:

Default Port	Description
27017	The default port for `mongod` and `mongos` instances. You can change this port with `port` or *`--port`*.
27018	The default port when running with *`--shardsvr`* runtime operation or the `shardsvr` value for the `clusterRole` setting in a configuration file.
27019	The default port when running with *`--configsvr`* runtime operation or the `configsvr` value for the `clusterRole` setting in a configuration file.
28017	The default port for the web status page. The web status page is always accessible at a port number that is `1000` greater than the port determined by `port`.

System Event Audit Messages

On this page

Note: Available only in MongoDB Enterprise[78].

Audit Message

The *event auditing feature* (page 325) can record events in JSON format. To configure auditing output, see *Configure System Events Auditing* (page 397)

The recorded JSON messages have the following syntax:

```
{
  atype: <String>,
  ts : { "$date": <timestamp> },
  local: { ip: <String>, port: <int> },
  remote: { ip: <String>, port: <int> },
  users : [ { user: <String>, db: <String> }, ... ],
  roles: [ { role: <String>, db: <String> }, ... ],
  param: <document>,
  result: <int>
}
```

[78] http://www.mongodb.com/products/mongodb-enterprise

field String atype Action type. See *Audit Event Actions, Details, and Results* (page 425).

field document ts Document that contains the date and UTC time of the event, in ISO 8601 format.

field document local Document that contains the local `ip` address and the `port` number of the running instance.

field document remote Document that contains the remote `ip` address and the `port` number of the incoming connection associated with the event.

field array users Array of user identification documents. Because MongoDB allows a session to log in with different user per database, this array can have more than one user. Each document contains a `user` field for the username and a `db` field for the authentication database for that user.

field array roles Array of documents that specify the *roles* (page 320) granted to the user. Each document contains a `role` field for the name of the role and a `db` field for the database associated with the role.

field document param Specific details for the event. See *Audit Event Actions, Details, and Results* (page 425).

field integer result Error code. See *Audit Event Actions, Details, and Results* (page 425).

Audit Event Actions, Details, and Results

The following table lists for each `atype` or action type, the associated `param` details and the `result` values, if any.

atype	param	result
`authenticate`	`{ user: <user name>, db: <database>, mechanism: <mechanism> }`	`0` - Success `18` - Authentication Failed
`authCheck`	`{ command: <name>, ns: <database>.<collection>, args: <command object> }` `ns` field is optional. `args` field may be redacted.	`0` - Success `13` - Unauthorized to perform the operation. By default, the auditing system logs only the authorization failures. To enable the system to log authorization successes, use the `auditAuthorizationSuccess` parameter. [79]
`createCollection` (page 419)	`{ ns: <database>.<collection> }`	`0` - Success
`createDatabase`	`{ ns: <database> }`	`0` - Success

Continued on next page

[79] Enabling `auditAuthorizationSuccess` degrades performance more than logging only the authorization failures.

Table 6.1 – continued from previous page

atype	param	result
createIndex (page 419)	{ ns: <database>.<collection>, indexName: <index name>, indexSpec: <index specification> }	0 - Success
renameCollection	{ old: <database>.<collection>, new: <database>.<collection> }	0 - Success
dropCollection (page 420)	{ ns: <database>.<collection> }	0 - Success
dropDatabase (page 422)	{ ns: <database> }	0 - Success
dropIndex (page 422)	{ ns: <database>.<collection>, indexName: <index name> }	0 - Success
createUser (page 420)	{ user: <user name>, db: <database>, customData: <document>, roles: [{ role: <role name>, db: <database> }, ...] } The customData field is optional.	0 - Success
dropUser (page 420)	{ user: <user name>, db: <database> }	0 - Success
dropAllUsersFromDatabase	{ db: <database> }	0 - Success

Continued on next page

Table 6.1 – continued from previous page

atype	param	result
updateUser	{ user: <user name>, db: <database>, passwordChanged: <boolean>, customData: <document>, roles: [{ role: <role name>, db: <database> }, ...] } The customData field is optional.	0 - Success
grantRolesToUser	{ user: <user name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...] }	0 - Success
revokeRolesFromUser	{ user: <user name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...] }	0 - Success

Continued on next page

Table 6.1 – continued from previous page

atype	param	result
createRole (page 420)	{ role: <role name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...], privileges: [{ resource: <resource document>, actions: [<action>, ...] }, ...] } The roles and the privileges fields are optional. For details on the resource document, see *Resource Document* (page 417). For a list of actions, see *Privilege Actions* (page 418).	0 - Success
updateRole	{ role: <role name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...], privileges: [{ resource: <resource document>, actions: [<action>, ...] }, ...] } The roles and the privileges fields are optional. For details on the resource document, see *Resource Document* (page 417). For a list of actions, see *Privilege Actions* (page 418).	0 - Success

Continued on next page

Table 6.1 – continued from previous page

atype	param	result
dropRole (page 420)	{ role: <role name>, db: <database> }	0 - Success
dropAllRolesFromDatabase	{ db: <database> }	0 - Success
grantRolesToRole	{ role: <role name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...] }	0 - Success
revokeRolesFromRole	{ role: <role name>, db: <database>, roles: [{ role: <role name>, db: <database> }, ...] }	0 - Success
grantPrivilegesToRole	{ role: <role name>, db: <database>, privileges: [{ resource: <resource document>, actions: [<action>, ...] }, ...] } For details on the resource document, see *Resource Document* (page 417). For a list of actions, see *Privilege Actions* (page 418).	0 - Success

Continued on next page

Table 6.1 – continued from previous page

atype	param	result
revokePrivilegesFromRole	{ role: <role name>, db: <database name>, privileges: [{ resource: <resource document>, actions: [<action>, ...] }, ...] } For details on the resource document, see *Resource Document* (page 417). For a list of actions, see *Privilege Actions* (page 418).	0 - Success
replSetReconfig	{ old: <configuration>, new: <configuration> } Indicates membership change in the replica set. The old field is optional.	0 - Success
enableSharding (page 421)	{ ns: <database> }	0 - Success
shardCollection	{ ns: <database>.<collection>, key: <shard key pattern>, options: { unique: <boolean> } }	0 - Success
addShard (page 421)	{ shard: <shard name>, connectionString: <hostname>:<port>, maxSize: <maxSize> } When a shard is a replica set, the connectionString includes the replica set name and can include other members of the replica set.	0 - Success
removeShard (page 422)	{ shard: <shard name> }	0 - Success
shutdown (page 423)	{ } Indicates commencement of database shutdown.	0 - Success

Continued on next page

Table 6.1 – continued from previous page

atype	param	result
applicationMessage (page 422)	{ msg: <custom message string> } See logApplicationMessage.	0 - Success

6.4.3 Security Release Notes Alerts

Security Release Notes **(page 431)** Security vulnerability for password.

Security Release Notes

On this page

Access to system.users Collection

Changed in version 2.4.

In 2.4, only users with the userAdmin role have access to the system.users collection.

In version 2.2 and earlier, the read-write users of a database all have access to the system.users collection, which contains the user names and user password hashes. [80]

Password Hashing Insecurity

If a user has the same password for multiple databases, the hash will be the same. A malicious user could exploit this to gain access on a second database using a different user's credentials.

As a result, always use unique username and password combinations for each database.

Thanks to Will Urbanski, from Dell SecureWorks, for identifying this issue.

6.5 Security Checklist

[80] Read-only users do not have access to the system.users collection.

On this page

This documents provides a list of security measures that you should implement to protect your MongoDB installation.

6.5.1 Require Authentication

Enable MongoDB authentication and specify the authentication mechanism. You can use the MongoDB authentication mechanism or an existing external framework. Authentication requires that all clients and servers provide valid credentials before they can connect to the system. In clustered deployments, enable authentication for each MongoDB server.

See *Authentication* (page 316), *Enable Client Access Control* (page 353), and *Enable Authentication in a Sharded Cluster* (page 354).

6.5.2 Configure Role-Based Access Control

Create roles that define the exact access a set of users needs. Follow a principle of least privilege. Then create users and assign them only the roles they need to perform their operations. A user can be a person or a client application.

Create a user administrator first, then create additional users. Create a unique MongoDB user for each person and application that accesses the system.

See *Authorization* (page 320), *Create a Role* (page 386), *Create a User Administrator* (page 381), and *Add a User to a Database* (page 383).

6.5.3 Encrypt Communication

Configure MongoDB to use TLS/SSL for all incoming and outgoing connections. Use TLS/SSL to encrypt communication between `mongod` and `mongos` components of a MongoDB client as well as between all applications and MongoDB.

See *Configure mongod and mongos for TLS/SSL* (page 338).

6.5.4 Limit Network Exposure

Ensure that MongoDB runs in a trusted network environment and limit the interfaces on which MongoDB instances listen for incoming connections. Allow only trusted clients to access the network interfaces and ports on which MongoDB instances are available.

See the `bindIp` setting, and see *Configure Linux iptables Firewall for MongoDB* (page 331) and *Configure Windows netsh Firewall for MongoDB* (page 334).

6.5.5 Audit System Activity

Track access and changes to database configurations and data. MongoDB Enterprise[81] includes a system auditing facility that can record system events (e.g. user operations, connection events) on a MongoDB instance. These audit records permit forensic analysis and allow administrators to verify proper controls.

See *Auditing* (page 325) and *Configure System Events Auditing* (page 397).

6.5.6 Encrypt and Protect Data

Encrypt MongoDB data on each host using file-system, device, or physical encryption. Protect MongoDB data using file-system permissions. MongoDB data includes data files, configuration files, auditing logs, and key files.

6.5.7 Run MongoDB with a Dedicated User

Run MongoDB processes with a dedicated operating system user account. Ensure that the account has permissions to access data but no unnecessary permissions.

See *Install MongoDB* (page 5) for more information on running MongoDB.

6.5.8 Run MongoDB with Secure Configuration Options

MongoDB supports the execution of JavaScript code for certain server-side operations: `mapReduce`, `group`, `eval`, and `$where`. If you do not use these operations, disable server-side scripting by using the `--noscripting` option on the command line.

Use only the MongoDB wire protocol on production deployments. Do **not** enable the following, all of which enable the web server interface: `enabled`, `net.http.JSONPEnabled`, and `net.http.RESTInterfaceEnabled`. Leave these *disabled*, unless required for backwards compatibility.

Keep input validation enabled. MongoDB enables input validation by default through the `wireObjectCheck` setting. This ensures that all documents stored by the `mongod` instance are valid *BSON*.

6.5.9 Request a Security Technical Implementation Guide (where applicable)

The Security Technical Implementation Guide (STIG) contains security guidelines for deployments within the United States Department of Defense. MongoDB Inc. provides its STIG, upon request, for situations where it is required. Please request a copy[82] for more information.

6.5.10 Consider Security Standards Compliance

For applications requiring HIPAA or PCI-DSS compliance, please refer to the MongoDB Security Reference Architecture[83] to learn more about how you can use the key security capabilities to build compliant application infrastructure.

[81] http://www.mongodb.com/products/mongodb-enterprise
[82] http://www.mongodb.com/lp/contact/stig-requests
[83] http://info.mongodb.com/rs/mongodb/images/MongoDB_Security_Architecture_WP.pdf

CHAPTER 7

Aggregation

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the *aggregation pipeline* (page 439), the *map-reduce function* (page 442), and *single purpose aggregation methods and commands* (page 444).

Aggregation Introduction **(page 435)** A high-level introduction to aggregation.

Aggregation Concepts **(page 439)** Introduces the use and operation of the data aggregation modalities available in MongoDB.

- ***Aggregation Pipeline*** **(page 439)** The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.
- ***Map-Reduce*** **(page 442)** Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.
- ***Single Purpose Aggregation Operations*** **(page 444)** MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.
- ***Aggregation Mechanics*** **(page 447)** Details internal optimization operations, limits, support for sharded collections, and concurrency concerns.

Aggregation Examples **(page 453)** Examples and tutorials for data aggregation operations in MongoDB.

Aggregation Reference **(page 470)** References for all aggregation operations material for all data aggregation methods in MongoDB.

7.1 Aggregation Introduction

On this page

Aggregations are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the `mongod` instance simplifies application code and limits resource requirements.

Like queries, aggregation operations in MongoDB use *collections* of documents as an input and return results in the form of one or more documents.

7.1.1 Aggregation Modalities

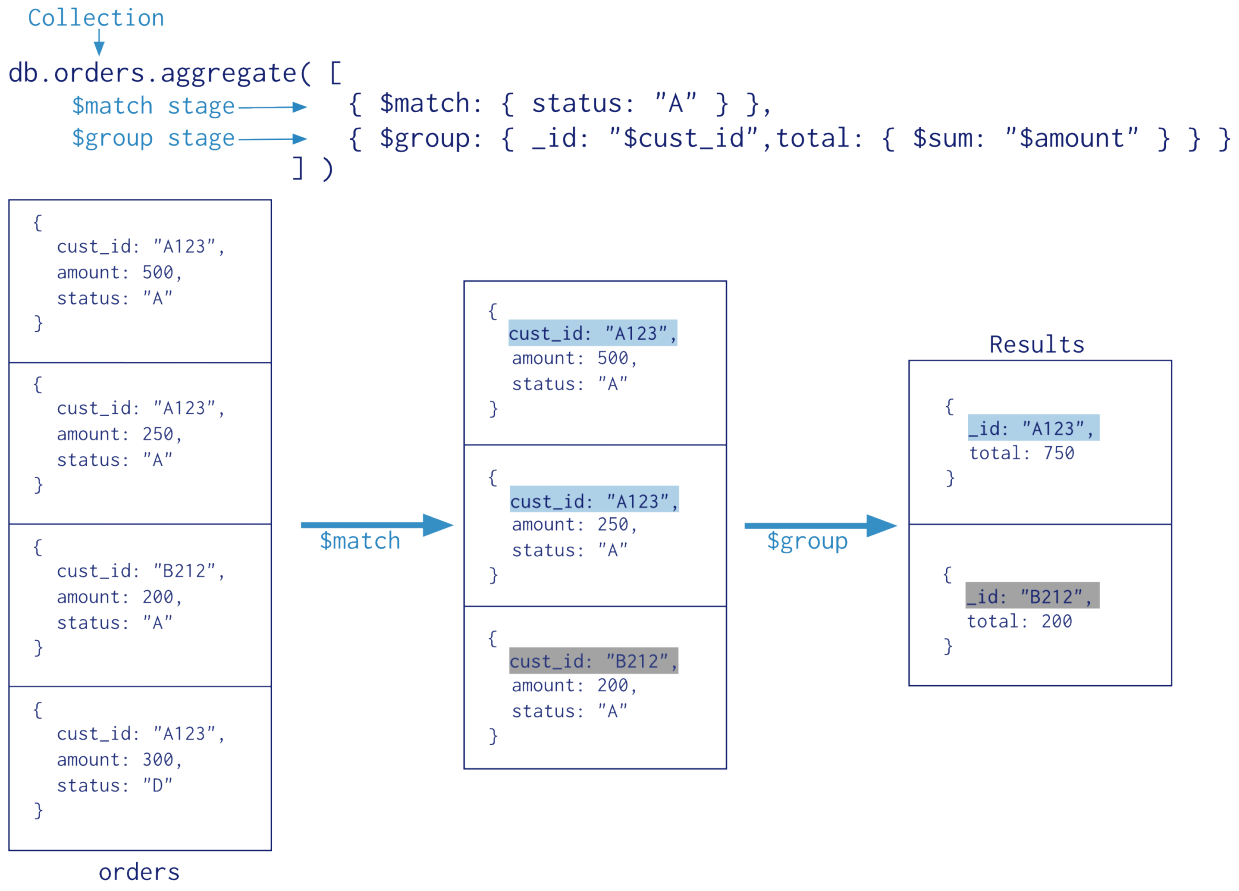
Aggregation Pipelines

MongoDB 2.2 introduced a new *aggregation framework* (page 439), modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

The most basic pipeline stages provide *filters* that operate like queries and *document transformations* that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use *operators* for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.



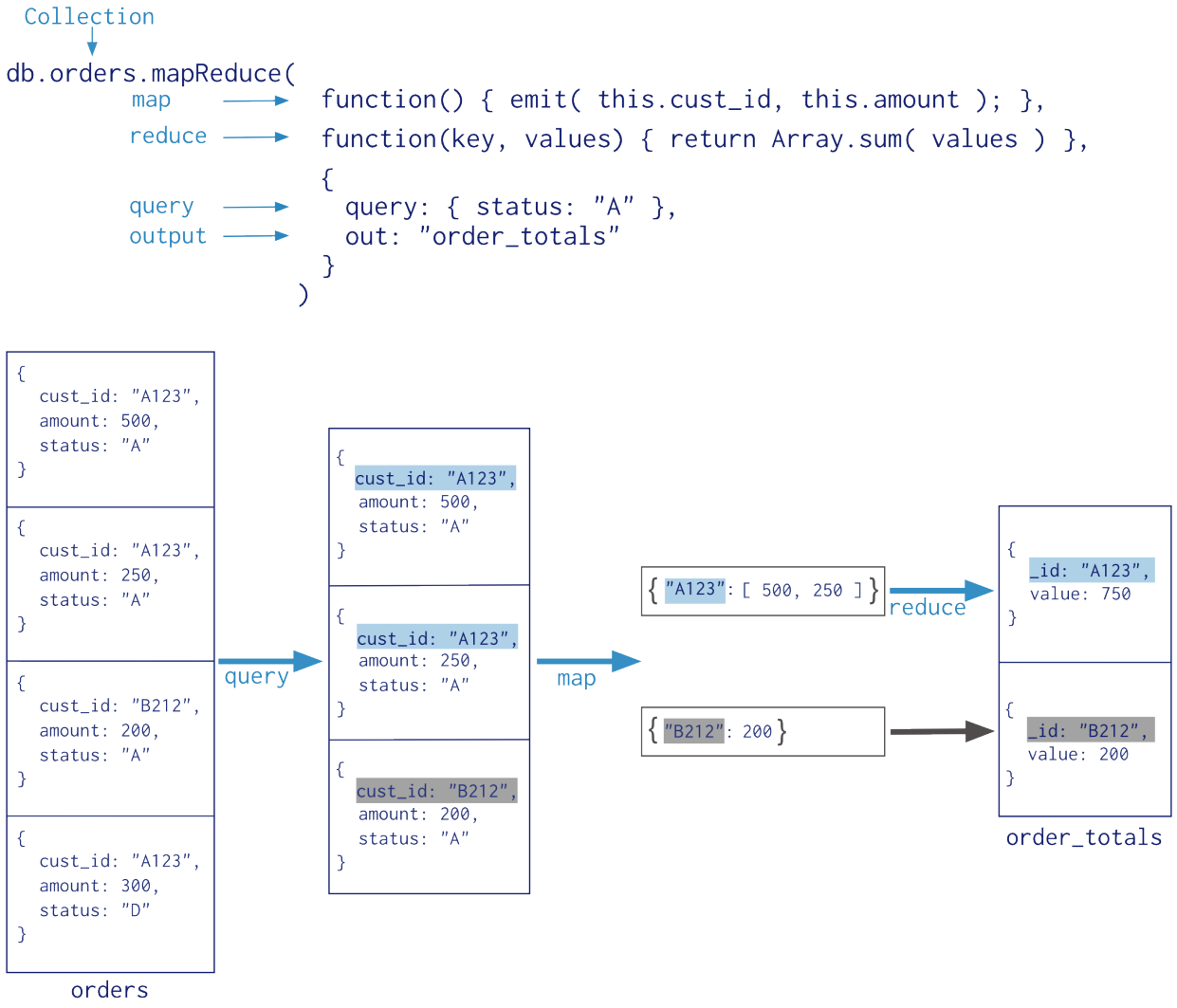
Map-Reduce

MongoDB also provides *map-reduce* (page 442) operations to perform aggregation. In general, map-reduce operations have two phases: a *map* stage that processes each document and *emits* one or more objects for each input document,

and *reduce* phase that combines the output of the map operation. Optionally, map-reduce can have a *finalize* stage to make final modifications to the result. Like other aggregation operations, map-reduce can specify a query condition to select the input documents as well as sort and limit the results.

Map-reduce uses custom JavaScript functions to perform the map and reduce operations, as well as the optional *finalize* operation. While the custom JavaScript provide great flexibility compared to the aggregation pipeline, in general, map-reduce is less efficient and more complex than the aggregation pipeline.

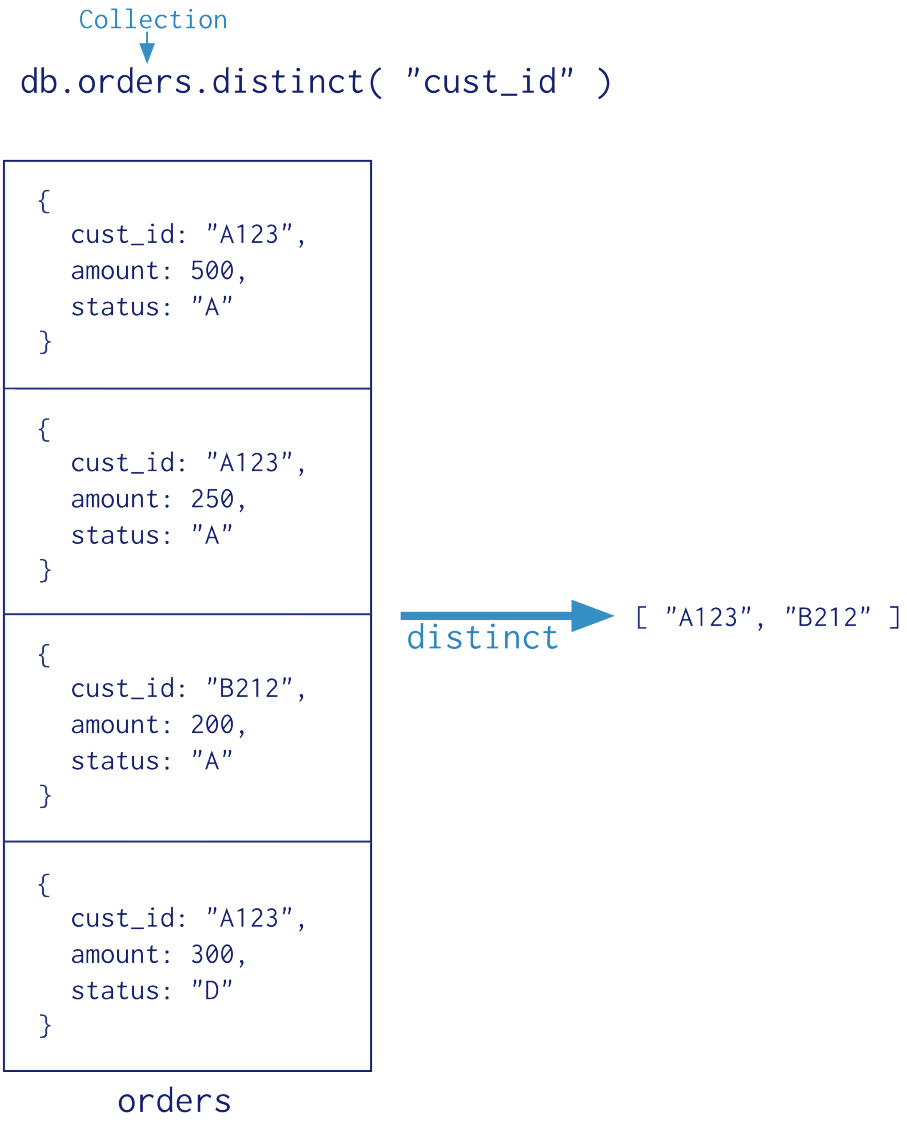
Note: Starting in MongoDB 2.4, certain `mongo` shell functions and properties are inaccessible in map-reduce operations. MongoDB 2.4 also provides support for multiple JavaScript operations to run at the same time. Before MongoDB 2.4, JavaScript code executed in a single thread, raising concurrency issues for map-reduce.



Single Purpose Aggregation Operations

For a number of common *single purpose aggregation operations* (page 444), MongoDB provides special purpose database commands. These common aggregation operations are: returning a count of matching documents, returning the distinct values for a field, and grouping data based on the values of a field. All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes,

they lack the flexibility and capabilities of the aggregation pipeline and map-reduce.



7.1.2 Additional Features and Behaviors

Both the aggregation pipeline and map-reduce can operate on a *sharded collection* (page 675). Map-reduce operations can also output to a sharded collection. See *Aggregation Pipeline and Sharded Collections* (page 451) and *Map-Reduce and Sharded Collections* (page 451) for details.

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase. See *Pipeline Operators and Indexes* (page 441) and *Aggregation Pipeline Optimization* (page 447) for details.

For a feature comparison of the aggregation pipeline, map-reduce, and the special group functionality, see *Aggregation Commands Comparison* (page 475).

7.1.3 Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data[1]
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop[2]
- The Aggregation Framework[3]

7.2 Aggregation Concepts

MongoDB provides the three approaches to aggregation, each with its own strengths and purposes for a given situation. This section describes these approaches and also describes behaviors and limitations specific to each approach. See also the *chart* (page 475) that compares the approaches.

Aggregation Pipeline **(page 439)** The aggregation pipeline is a framework for performing aggregation tasks, modeled on the concept of data processing pipelines. Using this framework, MongoDB passes the documents of a single collection through a pipeline. The pipeline transforms the documents into aggregated results, and is accessed through the `aggregate` database command.

Map-Reduce **(page 442)** Map-reduce is a generic multi-phase data aggregation modality for processing quantities of data. MongoDB provides map-reduce with the `mapReduce` database command.

Single Purpose Aggregation Operations **(page 444)** MongoDB provides a collection of specific data aggregation operations to support a number of common data aggregation functions. These operations include returning counts of documents, distinct values of a field, and simple grouping operations.

Aggregation Mechanics **(page 447)** Details internal optimization operations, limits, support for sharded collections, and concurrency concerns.

7.2.1 Aggregation Pipeline

New in version 2.2.

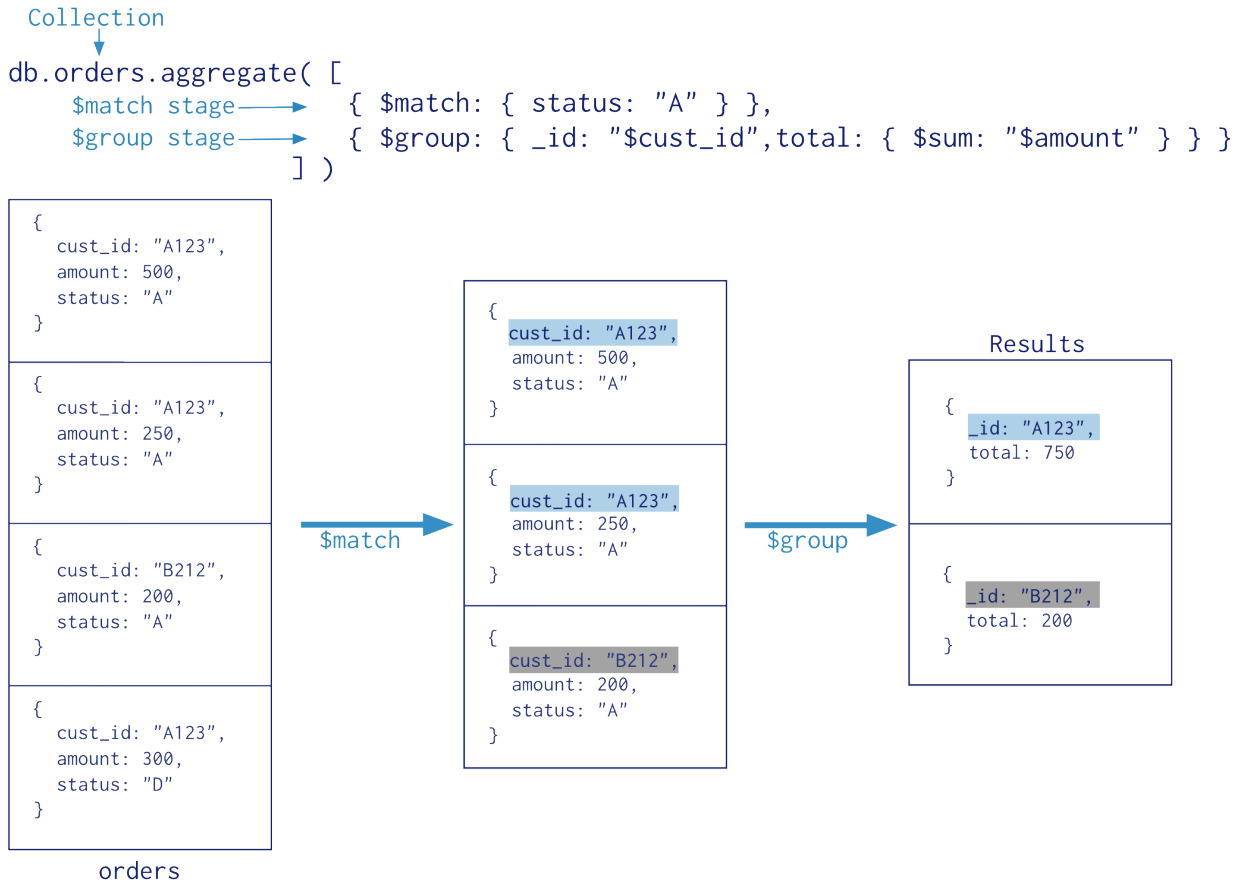
On this page

The aggregation pipeline is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for aggregation tasks where the complexity of map-reduce may be unwarranted.

Aggregation pipeline have some limitations on value types and result size. See *Aggregation Pipeline Limits* (page 450) for details on limits and restrictions on the aggregation pipeline.

[1] http://www.mongodb.com/presentations/mongodb-analytics-learn-aggregation-example-exploratory-analytics-and-visualization?jmp=docs
[2] http://www.mongodb.com/presentations/mongodb-time-series-data-part-2-analyzing-time-series-data-using-aggregation-framework?jmp=docs
[3] https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs


Collection
db.orders.aggregate([
$match stage
{ $match: { status: "A" } },
$group stage
{ $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
])
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

Pipeline

The MongoDB aggregation pipeline consists of *stages*. Each stage transforms the documents as they pass through the pipeline. Pipeline stages do not need to produce one output document for every input document; e.g., some stages may generate new documents or filter out documents. Pipeline stages can appear multiple times in the pipeline.

MongoDB provides the `db.collection.aggregate()` method in the `mongo` shell and the `aggregate` command for aggregation pipeline. See *aggregation-pipeline-operator-reference* for the available stages.

For example usage of the aggregation pipeline, consider *Aggregation with User Preference Data* (page 457) and *Aggregation with the Zip Code Data Set* (page 453).

Pipeline Expressions

Some pipeline stages takes a pipeline expression as its operand. Pipeline expressions specify the transformation to apply to the input documents. Expressions have a *document* (page 176) structure and can contain other *expression* (page 471).

Pipeline expressions can only operate on the current document in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents.

Generally, expressions are stateless and are only evaluated when seen by the aggregation process with one exception: *accumulator* expressions.

The accumulators, used with the `$group` pipeline operator, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline.

For more information on expressions, see *Expressions* (page 471).

Aggregation Pipeline Behavior

In MongoDB, the `aggregate` command operates on a single collection, logically passing the *entire* collection into the aggregation pipeline. To optimize the operation, wherever possible, use the following strategies to avoid scanning the entire collection.

Pipeline Operators and Indexes

The `$match` and `$sort` pipeline operators can take advantage of an index when they occur at the **beginning** of the pipeline.

New in version 2.4: The `$geoNear` pipeline operator takes advantage of a geospatial index. When using `$geoNear`, the `$geoNear` pipeline operation must appear as the first stage in an aggregation pipeline.

Even when the pipeline uses an index, aggregation still requires access to the actual documents; i.e. indexes cannot fully cover an aggregation pipeline.

Changed in version 2.6: In previous versions, for very select use cases, an index could cover a pipeline.

Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the `$match`, `$limit`, and `$skip` stages to restrict the documents that enter at the beginning of the pipeline. When placed at the beginning of a pipeline, `$match` operations use suitable indexes to scan only the matching documents in a collection.

Placing a `$match` pipeline stage followed by a `$sort` stage at the start of the pipeline is logically equivalent to a single query with a sort and can use an index. When possible, place `$match` operators at the beginning of the pipeline.

Additional Features

The aggregation pipeline has an internal optimization phase that provides improved performance for certain sequences of operators. For details, see *Aggregation Pipeline Optimization* (page 447).

The aggregation pipeline supports operations on sharded collections. See *Aggregation Pipeline and Sharded Collections* (page 451).

7.2.2 Map-Reduce

On this page

Map-reduce is a data processing paradigm for condensing large volumes of data into useful *aggregated* results. For map-reduce operations, MongoDB provides the `mapReduce` database command.

Consider the following map-reduce operation:

In this map-reduce operation, MongoDB applies the *map* phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the *reduce* phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a *finalize* function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the `mongod` process. Map-reduce operations take the documents of a single *collection* as the *input* and can perform any arbitrary sorting and limiting before beginning the map stage. `mapReduce` can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

Note: For most aggregation operations, the *Aggregation Pipeline* (page 439) provides better performance and more coherent interface. However, map-reduce operations provide some flexibility that is not presently available in the aggregation pipeline.

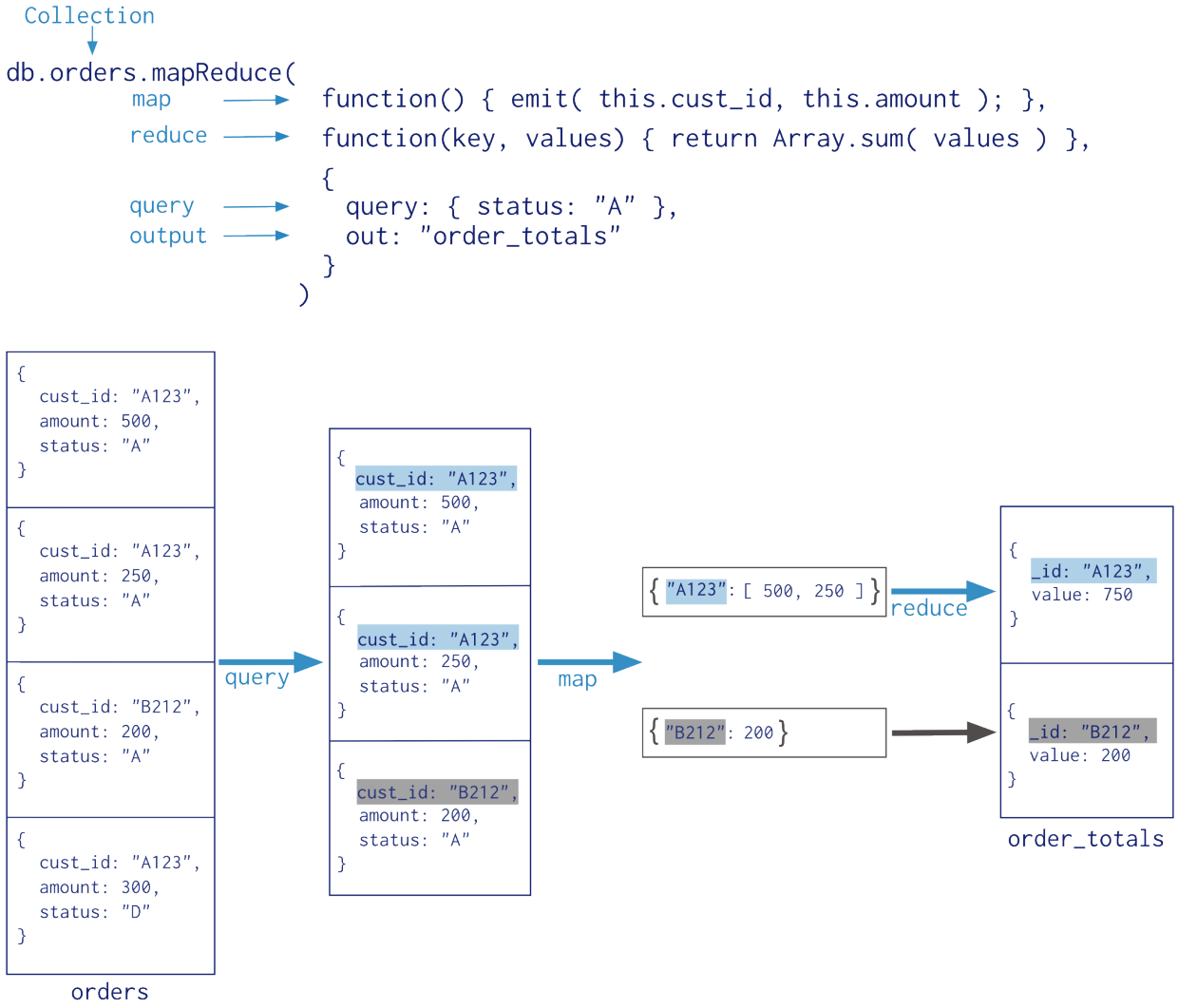
Map-Reduce JavaScript Functions

In MongoDB, map-reduce operations use custom JavaScript functions to *map*, or associate, values to a key. If a key has multiple values mapped to it, the operation *reduces* the values for the key to a single object.

The use of custom JavaScript functions provide flexibility to map-reduce operations. For instance, when processing a document, the map function can create more than one key and value mapping or no mapping. Map-reduce operations can also use a custom JavaScript function to make final modifications to the results at the end of the map and reduce operation, such as perform additional calculations.

Map-Reduce Behavior

In MongoDB, the map-reduce operation can write results to a collection or return the results inline. If you write map-reduce output to a collection, you can perform subsequent map-reduce operations on the same input collection that merge replace, merge, or reduce new results with previous results. See `mapReduce` and *Perform Incremental Map-Reduce* (page 464) for details and examples.


Collection
db.orders.mapReduce(
map
function() { emit(this.cust_id, this.amount); },
reduce
function(key, values) { return Array.sum(values) },
{
query
query: { status: "A" },
output
out: "order_totals"
}
)
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
query
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
map
{ "A123": [500, 250] }
{ "B212": 200 }
reduce
{ _id: "A123", value: 750 }
{ _id: "B212", value: 200 }
order_totals

When returning the results of a map reduce operation *inline*, the result documents must be within the `BSON Document Size` limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations, see the `http://docs.mongodb.org/manual/reference/command/mapReduce` reference page.

MongoDB supports map-reduce operations on *sharded collections* (page 675). Map-reduce operations can also output the results to a sharded collection. See *Map-Reduce and Sharded Collections* (page 451).

7.2.3 Single Purpose Aggregation Operations

On this page

Aggregation refers to a broad class of data manipulation operations that compute a result based on an input *and* a specific procedure. MongoDB provides a number of aggregation operations that perform specific aggregation operations on a set of data.

Although limited in scope, particularly compared to the *aggregation pipeline* (page 439) and *map-reduce* (page 442), these operations provide straightforward semantics for common data processing options.

Count

MongoDB can return a count of the number of documents that match a query. The `count` command as well as the `count()` and `cursor.count()` methods provide access to counts in the `mongo` shell.

Example

Given a collection named `records` with *only* the following documents:

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
```

The following operation would count all documents in the collection and return the number `4`:

```
db.records.count()
```

The following operation will count only the documents where the value of the field `a` is `1` and return `3`:

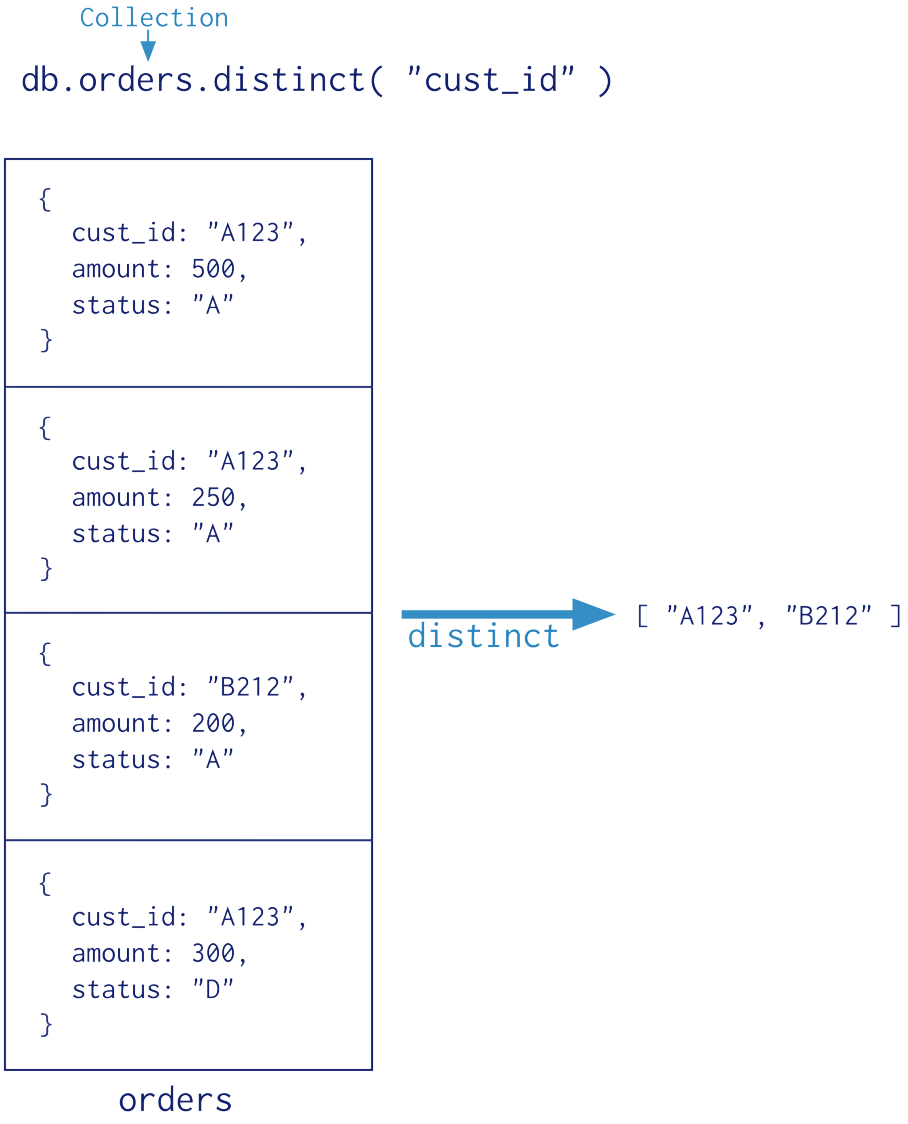
```
db.records.count( { a: 1 } )
```

Distinct

The *distinct* operation takes a number of documents that match a query and returns all of the unique values for a field in the matching documents. The `distinct` command and `db.collection.distinct()` method provide this operation in the `mongo` shell. Consider the following examples of a distinct operation:

Example

Given a collection named `records` with *only* the following documents:


Collection
db.orders.distinct("cust_id")
{
cust_id: "A123",
amount: 500,
status: "A"
}
{
cust_id: "A123",
amount: 250,
status: "A"
}
{
cust_id: "B212",
amount: 200,
status: "A"
}
{
cust_id: "A123",
amount: 300,
status: "D"
}
orders
distinct
["A123", "B212"]

```
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
{ a: 2, b: 2 }
{ a: 2, b: 2 }
```

Consider the following `db.collection.distinct()` operation which returns the distinct values of the field `b`:

```
db.records.distinct( "b" )
```

The results of this operation would resemble:

```
[ 0, 1, 4, 2 ]
```

Group

The *group* operation takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields. It returns an array of documents with computed results for each group of documents.

Access the grouping functionality via the `group` command or the `db.collection.group()` method in the `mongo` shell.

> **Warning:** `group` does not support data in sharded collections. In addition, the results of the `group` operation must be no larger than 16 megabytes.

Consider the following group operation:

Example

Given a collection named `records` with the following documents:

```
{ a: 1, count: 4 }
{ a: 1, count: 2 }
{ a: 1, count: 4 }
{ a: 2, count: 3 }
{ a: 2, count: 1 }
{ a: 1, count: 5 }
{ a: 4, count: 4 }
```

Consider the following `group` operation which groups documents by the field `a`, where `a` is less than `3`, and sums the field `count` for each group:

```
db.records.group( {
   key: { a: 1 },
   cond: { a: { $lt: 3 } },
   reduce: function(cur, result) { result.count += cur.count },
   initial: { count: 0 }
} )
```

The results of this group operation would resemble the following:

```
[
  { a: 1, count: 15 },
  { a: 2, count: 4 }
]
```

See also:

The `$group` for related functionality in the *aggregation pipeline* (page 439).

7.2.4 Aggregation Mechanics

This section describes behaviors and limitations for the various aggregation modalities.

Aggregation Pipeline Optimization **(page 447)** Details the internal optimization of certain pipeline sequence.

Aggregation Pipeline Limits **(page 450)** Presents limitations on aggregation pipeline operations.

Aggregation Pipeline and Sharded Collections **(page 451)** Mechanics of aggregation pipeline operations on sharded collections.

Map-Reduce and Sharded Collections **(page 451)** Mechanics of map-reduce operation with sharded collections.

Map Reduce Concurrency **(page 452)** Details the locks taken during map-reduce operations.

Aggregation Pipeline Optimization

On this page

Aggregation pipeline operations have an optimization phase which attempts to reshape the pipeline for improved performance.

To see how the optimizer transforms a particular aggregation pipeline, include the `explain` option in the `db.collection.aggregate()` method.

Optimizations are subject to change between releases.

Projection Optimization

The aggregation pipeline can determine if it requires only a subset of the fields in the documents to obtain the results. If so, the pipeline will only use those required fields, reducing the amount of data passing through the pipeline.

Pipeline Sequence Optimization

`$sort` + `$match` Sequence Optimization When you have a sequence with `$sort` followed by a `$match`, the `$match` moves before the `$sort` to minimize the number of objects to sort. For example, if the pipeline consists of the following stages:

```
{ $sort: { age : -1 } },
{ $match: { status: 'A' } }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $match: { status: 'A' } },
{ $sort: { age : -1 } }
```

$skip + $limit Sequence Optimization When you have a sequence with $skip followed by a $limit, the $limit moves before the $skip. With the reordering, the $limit value increases by the $skip amount.

For example, if the pipeline consists of the following stages:

```
{ $skip: 10 },
{ $limit: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```
{ $limit: 15 },
{ $skip: 10 }
```

This optimization allows for more opportunities for *$sort + $limit Coalescence* (page 448), such as with $sort + $skip + $limit sequences. See *$sort + $limit Coalescence* (page 448) for details on the coalescence and *$sort + $skip + $limit Sequence* (page 449) for an example.

For aggregation operations on *sharded collections* (page 451), this optimization reduces the results returned from each shard.

$redact + $match Sequence Optimization When possible, when the pipeline has the $redact stage immediately followed by the $match stage, the aggregation can sometimes add a portion of the $match stage before the $redact stage. If the added $match stage is at the start of a pipeline, the aggregation can use an index as well as query the collection to limit the number of documents that enter the pipeline. See *Pipeline Operators and Indexes* (page 441) for more information.

For example, if the pipeline consists of the following stages:

```
{ $redact: { $cond: { if: { $eq: [ "$level", 5 ] }, then: "$$PRUNE", else: "$$DESCEND" } } },
{ $match: { year: 2014, category: { $ne: "Z" } } }
```

The optimizer can add the same $match stage before the $redact stage:

```
{ $match: { year: 2014 } },
{ $redact: { $cond: { if: { $eq: [ "$level", 5 ] }, then: "$$PRUNE", else: "$$DESCEND" } } },
{ $match: { year: 2014, category: { $ne: "Z" } } }
```

Pipeline Coalescence Optimization

When possible, the optimization phase coalesces a pipeline stage into its predecessor. Generally, coalescence occurs *after* any sequence reordering optimization.

$sort + $limit Coalescence When a $sort immediately precedes a $limit, the optimizer can coalesce the $limit into the $sort. This allows the sort operation to only maintain the top n results as it progresses, where n is the specified limit, and MongoDB only needs to store n items in memory [4]. See *sort-and-memory* for more information.

[4] The optimization will still apply when allowDiskUse is true and the n items exceed the *aggregation memory limit* (page 451).

`$limit` + `$limit` Coalescence When a `$limit` immediately follows another `$limit`, the two stages can coalesce into a single `$limit` where the limit amount is the *smaller* of the two initial limit amounts. For example, a pipeline contains the following sequence:

```
{ $limit: 100 },
{ $limit: 10 }
```

Then the second `$limit` stage can coalesce into the first `$limit` stage and result in a single `$limit` stage where the limit amount `10` is the minimum of the two initial limits `100` and `10`.

```
{ $limit: 10 }
```

`$skip` + `$skip` Coalescence When a `$skip` immediately follows another `$skip`, the two stages can coalesce into a single `$skip` where the skip amount is the *sum* of the two initial skip amounts. For example, a pipeline contains the following sequence:

```
{ $skip: 5 },
{ $skip: 2 }
```

Then the second `$skip` stage can coalesce into the first `$skip` stage and result in a single `$skip` stage where the skip amount `7` is the sum of the two initial limits `5` and `2`.

```
{ $skip: 7 }
```

`$match` + `$match` Coalescence When a `$match` immediately follows another `$match`, the two stages can coalesce into a single `$match` combining the conditions with an `$and`. For example, a pipeline contains the following sequence:

```
{ $match: { year: 2014 } },
{ $match: { status: "A" } }
```

Then the second `$match` stage can coalesce into the first `$match` stage and result in a single `$match` stage

```
{ $match: { $and: [ { "year" : 2014 }, { "status" : "A" } ] } }
```

Examples

The following examples are some sequences that can take advantage of both sequence reordering and coalescence. Generally, coalescence occurs *after* any sequence reordering optimization.

`$sort` + `$skip` + `$limit` Sequence A pipeline contains a sequence of `$sort` followed by a `$skip` followed by a `$limit`:

```
{ $sort: { age : -1 } },
{ $skip: 10 },
{ $limit: 5 }
```

First, the optimizer performs the *$skip + $limit Sequence Optimization* (page 448) to transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $limit: 15 }
{ $skip: 10 }
```

The *$skip + $limit Sequence Optimization* (page 448) increases the `$limit` amount with the reordering. See *$skip + $limit Sequence Optimization* (page 448) for details.

The reordered sequence now has `$sort` immediately preceding the `$limit`, and the pipeline can coalesce the two stages to decrease memory usage during the sort operation. See *$sort + $limit Coalescence* (page 448) for more information.

`$limit` + `$skip` + `$limit` + `$skip` Sequence A pipeline contains a sequence of alternating `$limit` and `$skip` stages:

```
{ $limit: 100 },
{ $skip: 5 },
{ $limit: 10 },
{ $skip: 2 }
```

The *$skip + $limit Sequence Optimization* (page 448) reverses the position of the `{ $skip: 5 }` and `{ $limit: 10 }` stages and increases the limit amount:

```
{ $limit: 100 },
{ $limit: 15},
{ $skip: 5 },
{ $skip: 2 }
```

The optimizer then coalesces the two `$limit` stages into a single `$limit` stage and the two `$skip` stages into a single `$skip` stage. The resulting sequence is the following:

```
{ $limit: 15 },
{ $skip: 7 }
```

See *$limit + $limit Coalescence* (page 449) and *$skip + $skip Coalescence* (page 449) for details.

See also:

`explain` option in the `db.collection.aggregate()`

Aggregation Pipeline Limits

On this page

Aggregation operations with the `aggregate` command have the following limitations.

Result Size Restrictions

If the `aggregate` command returns a single document that contains the complete result set, the command will produce an error if the result set exceeds the `BSON Document Size` limit, which is currently 16 megabytes. To manage result sets that exceed this limit, the `aggregate` command can return result sets of *any size* if the command return a cursor or store the results to a collection.

Changed in version 2.6: The `aggregate` command can return results as a cursor or store the results in a collection, which are not subject to the size limit. The `db.collection.aggregate()` returns a cursor and can return result sets of any size.

Memory Restrictions

Changed in version 2.6.

Pipeline stages have a limit of 100 megabytes of RAM. If a stage exceeds this limit, MongoDB will produce an error. To allow for the handling of large datasets, use the `allowDiskUse` option to enable aggregation pipeline stages to write data to temporary files.

See also:

sort-memory-limit and *group-memory-limit*.

Aggregation Pipeline and Sharded Collections

On this page

The aggregation pipeline supports operations on *sharded* collections. This section describes behaviors specific to the *aggregation pipeline* (page 441) and sharded collections.

Behavior

Changed in version 2.6.

When operating on a sharded collection, the aggregation pipeline is split into two parts. The first pipeline runs on each shard, or if an early `$match` can exclude shards through the use of the shard key in the predicate, the pipeline runs on only the relevant shards.

The second pipeline consists of the remaining pipeline stages and runs on the *primary shard* (page 683). The primary shard merges the cursors from the other shards and runs the second pipeline on these results. The primary shard forwards the final results to the `mongos`. In previous versions, the second pipeline would run on the `mongos`. [5]

Optimization

When splitting the aggregation pipeline into two parts, the pipeline is split to ensure that the shards perform as many stages as possible with consideration for optimization.

To see how the pipeline was split, include the `explain` option in the `db.collection.aggregate()` method.

Optimizations are subject to change between releases.

Map-Reduce and Sharded Collections

On this page

[5] Until all shards upgrade to v2.6, the second pipeline runs on the `mongos` if any shards are still running v2.4.

Map-reduce supports operations on sharded collections, both as an input and as an output. This section describes the behaviors of `mapReduce` specific to sharded collections.

Sharded Collection as Input

When using sharded collection as the input for a map-reduce operation, `mongos` will automatically dispatch the map-reduce job to each shard in parallel. There is no special option required. `mongos` will wait for jobs on all shards to finish.

Sharded Collection as Output

Changed in version 2.2.

If the `out` field for `mapReduce` has the `sharded` value, MongoDB shards the output collection using the `_id` field as the shard key.

To output to a sharded collection:

- If the output collection does not exist, MongoDB creates and shards the collection on the `_id` field.
- For a new or an empty sharded collection, MongoDB uses the results of the first stage of the map-reduce operation to create the initial *chunks* distributed among the shards.
- `mongos` dispatches, in parallel, a map-reduce post-processing job to every shard that owns a chunk. During the post-processing, each shard will pull the results for its own chunks from the other shards, run the final reduce/finalize, and write locally to the output collection.

Note:

- During later map-reduce jobs, MongoDB splits chunks as needed.
- Balancing of chunks for the output collection is automatically prevented during post-processing to avoid concurrency issues.

In MongoDB 2.0:

- `mongos` retrieves the results from each shard, performs a merge sort to order the results, and proceeds to the reduce/finalize phase as needed. `mongos` then writes the result to the output collection in sharded mode.
- This model requires only a small amount of memory, even for large data sets.
- Shard chunks are not automatically split during insertion. This requires manual intervention until the chunks are granular and balanced.

Important: For best results, only use the sharded output options for `mapReduce` in version 2.2 or later.

Map Reduce Concurrency

The map-reduce operation is composed of many tasks, including reads from the input collection, executions of the `map` function, executions of the `reduce` function, writes to a temporary collection during processing, and writes to the output collection.

During the operation, map-reduce takes the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The insert into the temporary collection takes a write lock for a single write.

- If the output collection does not exist, the creation of the output collection takes a write lock.
- If the output collection exists, then the output actions (i.e. `merge`, `replace`, `reduce`) take a write lock. This write lock is *global*, and blocks all operations on the `mongod` instance.

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, which releases the lock between writing each output document. See the `db.collection.mapReduce()` reference for more information.

7.3 Aggregation Examples

This document provides the practical examples that display the capabilities of *aggregation* (page 439).

Aggregation with the Zip Code Data Set **(page 453)** Use the aggregation pipeline to group values and to calculate aggregated sums and averages for a collection of United States zip codes.

Aggregation with User Preference Data **(page 457)** Use the pipeline to sort, normalize, and sum data on a collection of user data.

Map-Reduce Examples **(page 461)** Define map-reduce operations that select ranges, group data, and calculate sums and averages.

Perform Incremental Map-Reduce **(page 464)** Run a map-reduce operations over one collection and output results to another collection.

Troubleshoot the Map Function **(page 466)** Steps to troubleshoot the `map` function.

Troubleshoot the Reduce Function **(page 467)** Steps to troubleshoot the `reduce` function.

7.3.1 Aggregation with the Zip Code Data Set

On this page

The examples in this document use the `zipcodes` collection. This collection is available at: media.mongodb.org/zips.json[6]. Use `mongoimport` to load this data set into your `mongod` instance.

Data Model

Each document in the `zipcodes` collection has the following form:

[6] http://media.mongodb.org/zips.json

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

- The `_id` field holds the zip code as a string.
- The `city` field holds the city name. A city can have more than one zip code associated with it as different sections of the city can each have a different zip code.
- 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.

`aggregate()` Method

All of the following examples use the `aggregate()` helper in the `mongo` shell.

The `aggregate()` method uses the *aggregation pipeline* (page 441) to processes documents into aggregated results. An *aggregation pipeline* (page 441) consists of *stages* with each stage processing the documents as they pass along the pipeline. Documents pass through the stages in sequence.

The `aggregate()` method in the `mongo` shell provides a wrapper around the `aggregate` database command. See the documentation for your `driver` for a more idiomatic interface for data aggregation operations.

Return States with Populations above 10 Million

The following aggregation operation returns all states with total population greater than 10 million:

```
db.zipcodes.aggregate( [
   { $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
   { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

In this example, the *aggregation pipeline* (page 441) consists of the `$group` stage followed by the `$match` stage:

- The `$group` stage groups the documents of the `zipcode` collection by the `state` field, calculates the `totalPop` field for each state, and outputs a document for each unique state.

  The new per-state documents have two fields: the `_id` field and the `totalPop` field. The `_id` field contains the value of the `state`; i.e. the group by field. The `totalPop` field is a calculated field that contains the total population of each state. To calculate the value, `$group` uses the `$sum` operator to add the population field (`pop`) for each state.

  After the `$group` stage, the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```

- The `$match` stage filters these grouped documents to output only those documents whose `totalPop` value is greater than or equal to 10 million. The `$match` stage does not alter the matching documents but outputs the matching documents unmodified.

The equivalent *SQL* for this aggregation operation is:

```
SELECT state, SUM(pop) AS totalPop
FROM zipcodes
GROUP BY state
HAVING totalPop >= (10*1000*1000)
```

See also:

`$group`, `$match`, `$sum`

Return Average City Population by State

The following aggregation operation returns the average populations for cities in each state:

```
db.zipcodes.aggregate( [
   { $group: { _id: { state: "$state", city: "$city" }, pop: { $sum: "$pop" } } },
   { $group: { _id: "$_id.state", avgCityPop: { $avg: "$pop" } } }
] )
```

In this example, the *aggregation pipeline* (page 441) consists of the `$group` stage followed by another `$group` stage:

- The first `$group` stage groups the documents by the combination of `city` and `state`, uses the `$sum` expression to calculate the population for each combination, and outputs a document for each `city` and `state` combination. [7]

  After this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- A second `$group` stage groups the documents in the pipeline by the `_id.state` field (i.e. the `state` field inside the `_id` document), uses the `$avg` expression to calculate the average city population (`avgCityPop`) for each state, and outputs a document for each state.

The documents that result from this aggregation operation resembles the following:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
}
```

See also:

`$group`, `$sum`, `$avg`

[7] A city can have more than one zip code associated with it as different sections of the city can each have a different zip code.

Return Largest and Smallest Cities by State

The following aggregation operation returns the smallest and largest cities by population for each state:

```
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" }
    }
  }
] )
```

In this example, the *aggregation pipeline* (page 441) consists of a `$group` stage, a `$sort` stage, another `$group` stage, and a `$project` stage:

- The first `$group` stage groups the documents by the combination of the `city` and `state`, calculates the `sum` of the `pop` values for each combination, and outputs a document for each `city` and `state` combination.

  At this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- The `$sort` stage orders the documents in the pipeline by the `pop` field value, from smallest to largest; i.e. by increasing order. This operation does not alter the documents.

- The next `$group` stage groups the now-sorted documents by the `_id.state` field (i.e. the `state` field inside the `_id` document) and outputs a document for each state.

  The stage also calculates the following four fields for each state. Using the `$last` expression, the `$group` operator creates the `biggestCity` and `biggestPop` fields that store the city with the largest population and that population. Using the `$first` expression, the `$group` operator creates the `smallestCity` and `smallestPop` fields that store the city with the smallest population and that population.

The documents, at this stage in the pipeline, resemble the following:

```
{
  "_id" : "WA",
  "biggestCity" : "SEATTLE",
  "biggestPop" : 520096,
  "smallestCity" : "BENGE",
  "smallestPop" : 2
}
```

- The final `$project` stage renames the `_id` field to `state` and moves the `biggestCity`, `biggestPop`, `smallestCity`, and `smallestPop` into `biggestCity` and `smallestCity` embedded documents.

The output documents of this aggregation operation resemble the following:

```
{
  "state" : "RI",
  "biggestCity" : {
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
    "pop" : 45
  }
}
```

7.3.2 Aggregation with User Preference Data

On this page

Data Model

Consider a hypothetical sports club with a database that contains a `users` collection that tracks the user's join dates, sport preferences, and stores these data in documents that resemble the following:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
{
  _id : "joe",
  joined : ISODate("2012-07-02"),
  likes : ["tennis", "golf", "swimming"]
}
```

Normalize and Sort Documents

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the `users` collection. You might do this to normalize user names for processing.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

All documents from the `users` collection pass through the pipeline, which consists of the following operations:

- The `$project` operator:
  - creates a new field called `name`.
  - converts the value of the `_id` to upper case, with the `$toUpper` operator. Then the `$project` creates a new field, named `name` to hold this value.
  - suppresses the `id` field. `$project` will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

Return Usernames Ordered by Join Month

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

```
db.users.aggregate(
  [
    { $project :
       {
         month_joined : { $month : "$joined" },
         name : "$_id",
         _id : 0
       }
    },
    { $sort : { month_joined : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator:
  - Creates two new fields: `month_joined` and `name`.

  - Suppresses the `id` from the results. The `aggregate()` method includes the `_id`, unless explicitly suppressed.

- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns those values to the `month_joined` field.
- The `$sort` operator sorts the results by the `month_joined` field.

The operation returns results that resemble the following:

```
{
  "month_joined" : 1,
  "name" : "ruth"
},
{
  "month_joined" : 1,
  "name" : "harold"
},
{
  "month_joined" : 1,
  "name" : "kate"
}
{
  "month_joined" : 2,
  "name" : "jill"
}
```

Return Total Number of Joins per Month

The following operation shows how many people joined each month of the year. You might use this aggregated data for recruiting and marketing strategies.

```
db.users.aggregate(
  [
    { $project : { month_joined : { $month : "$joined" } } } ,
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum : 1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator creates a new field called `month_joined`.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns the values to the `month_joined` field.
- The `$group` operator collects all documents with a given `month_joined` value and counts how many documents there are for that value. Specifically, for each unique value, `$group` creates a new "per-month" document with two fields:
  - `_id`, which contains a nested document with the `month_joined` field and its value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` operator sorts the documents created by `$group` according to the contents of the `month_joined` field.

The result of this aggregation operation would resemble the following:

```
{
  "_id" : {
    "month_joined" : 1
  },
  "number" : 3
},
{
  "_id" : {
    "month_joined" : 2
  },
  "number" : 9
},
{
  "_id" : {
    "month_joined" : 3
  },
  "number" : 5
}
```

Return the Five Most Common "Likes"

The following aggregation collects top five most "liked" activities in the data set. This type of analysis could help inform planning and future development.

```
db.users.aggregate(
  [
    { $unwind : "$likes" },
    { $group : { _id : "$likes" , number : { $sum : 1 } } },
    { $sort : { number : -1 } },
    { $limit : 5 }
  ]
)
```

The pipeline begins with all documents in the `users` collection, and passes these documents through the following operations:

- The `$unwind` operator separates each value in the `likes` array, and creates a new version of the source document for every element in the array.

  Example

  Given the following document from the `users` collection:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : ["golf", "racquetball"]
  }
  ```

  The `$unwind` operator would create the following documents:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : "golf"
  }
  {
    _id : "jane",
  ```

```
    joined : ISODate("2011-03-02"),
    likes : "racquetball"
}
```

- The `$group` operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` creates a new document with two fields:
  - `_id`, which contains the `likes` value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` operator sorts these documents by the `number` field in reverse order.
- The `$limit` operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

```
{
  "_id" : "golf",
  "number" : 33
},
{
  "_id" : "racquetball",
  "number" : 31
},
{
  "_id" : "swimming",
  "number" : 24
},
{
  "_id" : "handball",
  "number" : 19
},
{
  "_id" : "tennis",
  "number" : 18
}
```

7.3.3 Map-Reduce Examples

On this page

In the `mongo` shell, the `db.collection.mapReduce()` method is a wrapper around the `mapReduce` command. The following examples use the `db.collection.mapReduce()` method:

Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
```

```
    status: 'A',
    price: 25,
    items: [ { sku: "mmm", qty: 5, price: 2.5 },
             { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer

Perform the map-reduce operation on the `orders` collection to group by the `cust_id`, and calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

   ```
   var mapFunction1 = function() {
                          emit(this.cust_id, this.price);
                      };
   ```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:
   - The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
   - The function reduces the `valuesPrice` array to the sum of its elements.

   ```
   var reduceFunction1 = function(keyCustId, valuesPrices) {
                             return Array.sum(valuesPrices);
                         };
   ```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

   ```
   db.orders.mapReduce(
                       mapFunction1,
                       reduceFunction1,
                       { out: "map_reduce_example" }
                     )
   ```

   This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate Order and Total Quantity with Average Quantity Per Item

In this example, you will perform a map-reduce operation on the `orders` collection for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and calculates the number of orders and the total quantity ordered for each `sku`. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
                                         qty: this.items[idx].qty
                                       };
                           emit(key, value);
                       }
                   };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `countObjVals`:

   - `countObjVals` is an array whose elements are the objects mapped to the grouped `keySKU` values passed by map function to the reducer function.
   - The function reduces the `countObjVals` array to a single object `reducedValue` that contains the `count` and the `qty` fields.
   - In `reducedVal`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, countObjVals) {
                       reducedVal = { count: 0, qty: 0 };

                       for (var idx = 0; idx < countObjVals.length; idx++) {
                           reducedVal.count += countObjVals[idx].count;
                           reducedVal.qty += countObjVals[idx].qty;
                       }

                       return reducedVal;
                   };
```

3. Define a finalize function with two arguments `key` and `reducedVal`. The function modifies the `reducedVal` object to add a computed field named `avg` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedVal) {

                         reducedVal.avg = reducedVal.qty/reducedVal.count;

                         return reducedVal;

                       };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                       reduceFunction2,
                       {
                         out: { merge: "map_reduce_example" },
                         query: { ord_date:
                                     { $gt: new Date('01/01/2012') }
                                },
                         finalize: finalizeFunction2
                       }
                     )
```

This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the

map_reduce_example collection already exists, the operation will merge the existing contents with the results of this map-reduce operation.

7.3.4 Perform Incremental Map-Reduce

On this page

Map-reduce operations can handle complex aggregation tasks. To perform map-reduce operations, MongoDB provides the mapReduce command and, in the mongo shell, the db.collection.mapReduce() wrapper method.

If the map-reduce data set is constantly growing, you may want to perform an incremental map-reduce rather than performing the map-reduce operation over the entire data set each time.

To perform incremental map-reduce:

1. Run a map-reduce job over the current collection and output the result to a separate collection.
2. When you have more data to process, run subsequent map-reduce job with:
   - the query parameter that specifies conditions that match *only* the new documents.
   - the out parameter that specifies the reduce action to merge the new results into the existing output collection.

Consider the following example where you schedule a map-reduce operation on a sessions collection to run at the end of each day.

Data Setup

The sessions collection contains documents that log users' sessions each day, for example:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 } );

db.sessions.save( { userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 } );
```

Initial Map-Reduce of Current Collection

Run the first map-reduce operation as follows:

1. Define the map function that maps the userid to an object that contains the fields userid, total_time, count, and avg_time:

```
var mapFunction = function() {
                      var key = this.userid;
                      var value = {
                                      userid: this.userid,
```

```
                                    total_time: this.length,
                                    count: 1,
                                    avg_time: 0
                                   };

                emit( key, value );
            };
```

2. Define the corresponding reduce function with two arguments `key` and `values` to calculate the total time and the count. The `key` corresponds to the `userid`, and the `values` is an array whose elements corresponds to the individual objects mapped to the `userid` in the `mapFunction`.

```
var reduceFunction = function(key, values) {

                var reducedObject = {
                                    userid: key,
                                    total_time: 0,
                                    count:0,
                                    avg_time:0
                                   };

                values.forEach( function(value) {
                                    reducedObject.total_time += value.total_time;
                                    reducedObject.count += value.count;
                                }
                              );
                return reducedObject;
            };
```

3. Define the finalize function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` document to add another field `average` and returns the modified document.

```
var finalizeFunction = function (key, reducedValue) {

                  if (reducedValue.count > 0)
                      reducedValue.avg_time = reducedValue.total_time / reducedValue.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: "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
                       }
                     );
```

7.3.5 Troubleshoot the Map Function

The `map` function is a JavaScript function that associates or "maps" a value with a key and emits the key and value pair during a *map-reduce* (page 442) operation.

To verify the `key` and `value` pairs emitted by the `map` function, write your own `emit` function.

Consider a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

1. Define the `map` function that maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair:

   ```
   var map = function() {
       emit(this.cust_id, this.price);
   };
   ```

2. Define the `emit` function to print the key and value:

   ```
   var emit = function(key, value) {
       print("emit");
       print("key: " + key + "  value: " + tojson(value));
   }
   ```

3. Invoke the `map` function with a single document from the `orders` collection:

   ```
   var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
   map.apply(myDoc);
   ```

4. Verify the key and value pair is as you expected.

```
emit
key: abc123 value:250
```

5. Invoke the `map` function with multiple documents from the `orders` collection:

```
var myCursor = db.orders.find( { cust_id: "abc123" } );

while (myCursor.hasNext()) {
    var doc = myCursor.next();
    print ("document _id= " + tojson(doc._id));
    map.apply(doc);
    print();
}
```

6. Verify the key and value pairs are as you expected.

See also:

The `map` function must meet various requirements. For a list of all the requirements for the `map` function, see `mapReduce`, or the `mongo` shell helper method `db.collection.mapReduce()`.

7.3.6 Troubleshoot the Reduce Function

On this page

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 442) operation. The `reduce` function must meet various requirements. This tutorial helps verify that the `reduce` function meets the following criteria:

- The `reduce` function must return an object whose *type* must be **identical** to the type of the `value` emitted by the `map` function.
- The order of the elements in the `valuesArray` should not affect the output of the `reduce` function.
- The `reduce` function must be *idempotent*.

For a list of all the requirements for the `reduce` function, see `mapReduce`, or the `mongo` shell helper method `db.collection.mapReduce()`.

Confirm Output Type

You can test that the `reduce` function returns a value that is the same type as the value emitted from the `map` function.

1. Define a `reduceFunction1` function that takes the arguments `keyCustId` and `valuesPrices`. `valuesPrices` is an array of integers:

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                          return Array.sum(valuesPrices);
                      };
```

2. Define a sample array of integers:

```
var myTestValues = [ 5, 5, 10 ];
```

3. Invoke the `reduceFunction1` with `myTestValues`:

```
reduceFunction1('myKey', myTestValues);
```

4. Verify the `reduceFunction1` returned an integer:

```
20
```

5. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

6. Define a sample array of documents:

```
var myTestObjects = [
                        { count: 1, qty: 5 },
                        { count: 2, qty: 10 },
                        { count: 3, qty: 15 }
                    ];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Insensitivity to the Order of Mapped Values

The `reduce` function takes a `key` and a `values` array as its argument. You can test that the result of the `reduce` function does not depend on the order of the elements in the `values` array.

1. Define a sample `values1` array and a sample `values2` array that only differ in the order of the array elements:

```
var values1 = [
                  { count: 1, qty: 5 },
                  { count: 2, qty: 10 },
                  { count: 3, qty: 15 }
                ];

var values2 = [
                  { count: 3, qty: 15 },
                  { count: 1, qty: 5 },
                  { count: 2, qty: 10 }
                ];
```

2. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

3. Invoke the `reduceFunction2` first with `values1` and then with `values2`:

```
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Reduce Function Idempotence

Because the map-reduce operation may call a `reduce` multiple times for the same key, and won't call a `reduce` for single instances of a key in the working set, the `reduce` function must return a value of the same type as the value emitted from the `map` function. You can test that the `reduce` function process "reduced" values without affecting the *final* value.

1. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

2. Define a sample key:

```
var myKey = 'myKey';
```

3. Define a sample `valuesIdempotent` array that contains an element that is a call to the `reduceFunction2` function:

```
var valuesIdempotent = [
                         { count: 1, qty: 5 },
                         { count: 2, qty: 10 },
                         reduceFunction2(myKey, [ { count:3, qty: 15 } ] )
                       ];
```

4. Define a sample `values1` array that combines the values passed to `reduceFunction2`:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];
```

5. Invoke the `reduceFunction2` first with `myKey` and `valuesIdempotent` and then with `myKey` and `values1`:

   ```
   reduceFunction2(myKey, valuesIdempotent);
   reduceFunction2(myKey, values1);
   ```

6. Verify the `reduceFunction2` returned the same result:

   ```
   { "count" : 6, "qty" : 30 }
   ```

7.4 Aggregation Reference

Aggregation Pipeline Quick Reference **(page 470)** Quick reference card for aggregation pipeline.

`http://docs.mongodb.org/manual/reference/operator/aggregation` Aggregation pipeline operations have a collection of operators available to define and manipulate documents in pipeline stages.

Aggregation Commands Comparison **(page 475)** A comparison of `group`, `mapReduce` and `aggregate` that explores the strengths and limitations of each aggregation modality.

SQL to Aggregation Mapping Chart **(page 477)** An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

Aggregation Commands **(page 479)** The reference for the data aggregation commands, which provide the interfaces to MongoDB's aggregation capability.

Variables in Aggregation Expressions **(page 479)** Use of variables in aggregation pipeline expressions.

7.4.1 Aggregation Pipeline Quick Reference

On this page

Stages

Pipeline stages appear in an array. Documents pass through the stages in sequence. All except the `$out` and `$geoNear` stages can appear multiple times in a pipeline.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$redact	Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of $project and $match. Can be used to implement field level redaction. For each input document, outputs either one or zero document.
$limit	Passes the first *n* documents unmodified to the pipeline where *n* is the specified limit. For each input document, outputs either one document (for the first *n* documents) or zero documents (after the first *n* documents).
$skip	Skips the first *n* documents where *n* is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first *n* documents) or one document (if after the first *n* documents).
$unwind	Deconstructs an array field from the input documents to output a document for *each* element. Each output document replaces the array with an element value. For each input document, outputs *n* documents where *n* is the number of array elements and can be zero for an empty array.
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.
$geoNear	Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match, $sort, and $limit for geospatial data. The output documents include an additional distance field and can include a location identifier field.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.

Expressions

Expressions can include *field paths and system variables* (page 471), *literals* (page 472), *expression objects* (page 472), and *expression operators* (page 472). Expressions can be nested.

Field Path and System Variables

Aggregation expressions use *field path* to access fields in the input documents. To specify a field path, use a string that prefixes with a dollar sign $ the field name or the dotted field name, if the field is in embedded document. For example, "$user" to specify the field path for the user field or "$user.name" to specify the field path to "user.name" field.

"$<field>" is equivalent to "$$CURRENT.<field>" where the CURRENT (page 480) is a system variable that defaults to the root of the current object in the most stages, unless stated otherwise in specific stages. CURRENT (page 480) can be rebound.

Along with the CURRENT (page 480) system variable, other *system variables* (page 480) are also available for use in expressions. To use user-defined variables, use $let and $map expressions. To access variables in expressions, use a string that prefixes the variable name with $$.

Literals

Literals can be of any type. However, MongoDB parses string literals that start with a dollar sign `$` as a path to a field and numeric/boolean literals in *expression objects* (page 472) as projection flags. To avoid parsing literals, use the `$literal` expression.

Expression Objects

Expression objects have the following form:

```
{ <field1>: <expression1>, ... }
```

If the expressions are numeric or boolean literals, MongoDB treats the literals as projection flags (e.g. `1` or `true` to include the field), valid only in the `$project` stage. To avoid treating numeric or boolean literals as projection flags, use the `$literal` expression to wrap the numeric or boolean literals.

Operator Expressions

Operator expressions are similar to functions that take arguments. In general, these expressions take an array of arguments and have the following form:

```
{ <operator>: [ <argument1>, <argument2> ... ] }
```

If operator accepts a single argument, you can omit the outer array designating the argument list:

```
{ <operator>: <argument> }
```

To avoid parsing ambiguity if the argument is a literal array, you must wrap the literal array in a `$literal` expression or keep the outer array that designates the argument list.

Boolean Expressions Boolean expressions evaluate their argument expressions as booleans and return a boolean as the result.

In addition to the `false` boolean value, Boolean expression evaluates as `false` the following: `null`, `0`, and `undefined` values. The Boolean expression evaluates all other values as `true`, including non-zero numeric values and arrays.

Name	Description
`$and`	Returns `true` only when *all* its expressions evaluate to `true`. Accepts any number of argument expressions.
`$or`	Returns `true` when *any* of its expressions evaluates to `true`. Accepts any number of argument expressions.
`$not`	Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.

Set Expressions Set expressions performs set operation on arrays, treating arrays as sets. Set expressions ignores the duplicate entries in each input array and the order of the elements.

If the set operation returns a set, the operation filters out duplicates in the result to output an array that contains only unique entries. The order of the elements in the output array is unspecified.

If a set contains a nested array element, the set expression does *not* descend into the nested array but evaluates the array at top-level.

Name	Description
$setEquals	Returns true if the input sets have the same distinct elements. Accepts two or more argument expressions.
$setIntersection	Returns a set with elements that appear in *all* of the input sets. Accepts any number of argument expressions.
$setUnion	Returns a set with elements that appear in *any* of the input sets. Accepts any number of argument expressions.
$setDifference	Returns a set with elements that appear in the first set but not in the second set; i.e. performs a relative complement[8] of the second set relative to the first. Accepts exactly two argument expressions.
$setIsSubset	Returns true if all elements of the first set appear in the second set, including when the first set equals the second set; i.e. not a strict subset[9]. Accepts exactly two argument expressions.
$anyElementTrue	Returns true if *any* elements of a set evaluate to true; otherwise, returns false. Accepts a single argument expression.
$allElementsTrue	Returns true if *no* element of a set evaluates to false, otherwise, returns false. Accepts a single argument expression.

Comparison Expressions Comparison expressions return a boolean except for $cmp which returns a number.

The comparison expressions take two argument expressions and compare both value and type, using the *specified BSON comparison order* (page 187) for values of different types.

Name	Description
$cmp	Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and −1 if the first value is less than the second.
$eq	Returns true if the values are equivalent.
$gt	Returns true if the first value is greater than the second.
$gte	Returns true if the first value is greater than or equal to the second.
$lt	Returns true if the first value is less than the second.
$lte	Returns true if the first value is less than or equal to the second.
$ne	Returns true if the values are *not* equivalent.

Arithmetic Expressions Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

Name	Description
$add	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
$subtract	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
$multiply	Multiplies numbers to return the product. Accepts any number of argument expressions.
$divide	Returns the result of dividing the first number by the second. Accepts two argument expressions.
$mod	Returns the remainder of the first number divided by the second. Accepts two argument expressions.

String Expressions String expressions, with the exception of $concat, only have a well-defined behavior for strings of ASCII characters.

$concat behavior is well-defined regardless of the characters used.

[8] http://en.wikipedia.org/wiki/Complement_(set_theory)
[9] http://en.wikipedia.org/wiki/Subset

Name	Description
$concat	Concatenates any number of strings.
$substr	Returns a substring of a string, starting at a specified index position up to a specified length. Accepts three expressions as arguments: the first argument must resolve to a string, and the second and third arguments must resolve to integers.
$toLower	Converts a string to lowercase. Accepts a single argument expression.
$toUpper	Converts a string to uppercase. Accepts a single argument expression.
$strcasecmp	Performs case-insensitive string comparison and returns: 0 if two strings are equivalent, 1 if the first string is greater than the second, and −1 if the first string is less than the second.

Text Search Expressions

Name	Description
$meta	Access text search metadata.

Array Expressions

Name	Description
$size	Returns the number of elements in the array. Accepts a single expression as argument.

Variable Expressions

Name	Description
$map	Applies a subexpression to each element of an array and returns the array of resulting values in orde Accepts named parameters.
$let	Defines variables for use within the scope of a subexpression and returns the result of the subexpress Accepts named parameters.

Literal Expressions

Name	Description
$literal	Return a value without parsing. Use for values that the aggregation pipeline may interpret as an expression. For example, use a $literal expression to a string that starts with a $ to avoid parsing a field path.

Date Expressions

Name	Description
$dayOfYear	Returns the day of the year for a date as a number between 1 and 366 (leap year).
$dayOfMonth	Returns the day of the month for a date as a number between 1 and 31.
$dayOfWeek	Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).
$year	Returns the year for a date as a number (e.g. 2014).
$month	Returns the month for a date as a number between 1 (January) and 12 (December).
$week	Returns the week number for a date as a number between 0 (the partial week that precedes the firs Sunday of the year) and 53 (leap year).
$hour	Returns the hour for a date as a number between 0 and 23.
$minute	Returns the minute for a date as a number between 0 and 59.
$second	Returns the seconds for a date as a number between 0 and 60 (leap seconds).
$millisecond	Returns the milliseconds of a date as a number between 0 and 999.

Conditional Expressions

Name	Description
$cond	A ternary operator that evaluates one expression, and depending on the result, returns the value o the other two expressions. Accepts either three expressions in an ordered list or three named par
$ifNull	Returns either the non-null result of the first expression or the result of the second expression if expression results in a null result. Null result encompasses instances of undefined values or miss fields. Accepts two expressions as arguments. The result of the second expression can be null.

Accumulators

Accumulators, available only for the `$group` stage, compute values by combining documents that share the same group key. Accumulators take as input a single expression, evaluating the expression once for each input document, and maintain their state for the group of documents.

Name	Description
`$sum`	Returns a sum for each group. Ignores non-numeric values.
`$avg`	Returns an average for each group. Ignores non-numeric values.
`$first`	Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.
`$last`	Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.
`$max`	Returns the highest expression value for each group.
`$min`	Returns the lowest expression value for each group.
`$push`	Returns an array of expression values for each group.
`$addToSet`	Returns an array of *unique* expression values for each group. Order of the array elements is undefined.

7.4.2 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.

	aggregate	mapReduce	group
Description	New in version 2.2. Designed with specific goals of improving performance and usability for aggregation tasks. Uses a "pipeline" approach where objects are transformed as they pass through a series of pipeline operators such as `$group`, `$match`, and `$sort`. See `http://docs.mongodb.org/manual/reference/operator/aggregation` for more information on the pipeline operators.	Implements the Map-Reduce aggregation for processing large data sets.	Provides grouping functionality. Is slower than the `aggregate` command and has less functionality than the `mapReduce` command.
Key Features	Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.	In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See *Map-Reduce Examples* (page 461) and *Perform Incremental Map-Reduce* (page 464).	Can either group by existing fields or with a custom `keyf` JavaScript function, can group by calculated fields. See `group` for information and example using the `keyf` function.
Flexibility	Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the `$project` pipeline operator. See `$project` for more information as well as `http://docs.mongodb.org/manual/reference/operator/aggregation` for more information on all the available pipeline operators.	Custom `map`, `reduce` and `finalize` JavaScript functions offer flexibility to aggregation logic. See `mapReduce` for details and restrictions on the functions.	Custom `reduce` and `finalize` JavaScript functions offer flexibility to grouping logic. See `group` for details and restrictions on these functions.
Output Results	Returns results in various options (inline as a document that contains the result set, a cursor to the result set) or stores the results in a collection. The result is subject to the *BSON Document size* limit if returned inline as a document that contains the result set. Changed in version 2.6: Can return results as a cursor or store the results to a collection.	Returns results in various options (inline, new collection, merge, replace, reduce). See `mapReduce` for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.	Returns results inline as an array of grouped items. The result set must fit within the *maximum BSON document size limit*. Changed in version 2.2: The returned array can contain at most 20,000 elements; i.e. at most 20,000 unique groupings. Previous versions had a limit of 10,000 elements.
Sharding Notes	Supports non-sharded and sharded input collections.	Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread.	Does **not** support sharded collection. Prior to 2.4, JavaScript code executed in a single thread.
More Information	See *Aggregation Pipeline* (page 439) and `aggregate`.	See *Map-Reduce* (page 442) and `mapReduce`.	See `group`.

7.4.3 SQL to Aggregation Mapping Chart

On this page

The *aggregation pipeline* (page 439) allows MongoDB to provide native aggregation capabilities that corresponds to many common data aggregation operations in SQL.

The following table provides an overview of common SQL aggregation terms, functions, and concepts and the corresponding MongoDB *aggregation operators*:

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	`$match`
GROUP BY	`$group`
HAVING	`$match`
SELECT	`$project`
ORDER BY	`$sort`
LIMIT	`$limit`
SUM()	`$sum`
COUNT()	`$sum`
join	No direct corresponding operator; *however*, the `$unwind` operator allows for somewhat similar functionality, but with fields embedded within the document.

Examples

The following table presents a quick reference of SQL aggregation statements and the corresponding MongoDB statements. The examples in the table assume the following conditions:

- The SQL examples assume *two* tables, `orders` and `order_lineitem` that join by the `order_lineitem.order_id` and the `orders.id` columns.
- The MongoDB examples assume *one* collection `orders` that contain documents of the following prototype:

```
{
  cust_id: "abc123",
  ord_date: ISODate("2012-11-02T17:04:11.102Z"),
  status: 'A',
  price: 50,
  items: [ { sku: "xxx", qty: 25, price: 1 },
           { sku: "yyy", qty: 25, price: 1 } ]
}
```

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: { month: { $month: "$ord_date" }, day: { $dayOfMonth: "$ord_date" }, year: { $year: "$ord_date"} } }, total: { $sum: "$price" } } } ] )`	For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.
`SELECT cust_id, count(*) FROM orders`	`db.orders.aggregate( [ { $group: {`	For `cust_id` with multiple records, return the `cust_id` and the corresponding record count.

Additional Resources

- MongoDB and MySQL Compared[10]
- Quick Reference Cards[11]
- MongoDB Database Modernization Consulting Package[12]

7.4.4 Aggregation Commands

On this page

Aggregation Commands

Name	Description
`aggregate`	Performs *aggregation tasks* (page 439) such as group using the aggregation framework.
`count`	Counts the number of documents in a collection.
`distinct`	Displays the distinct values found for a specified key in a collection.
`group`	Groups documents in a collection by the specified key and performs simple aggregation.
`mapReduce`	Performs *map-reduce* (page 442) aggregation for large data sets.

Aggregation Methods

Name	Description
`db.collection.aggregate()`	Provides access to the *aggregation pipeline* (page 439).
`db.collection.group()`	Groups documents in a collection by the specified key and performs simple aggregation.
`db.collection.mapReduce()`	Performs *map-reduce* (page 442) aggregation for large data sets.

7.4.5 Variables in Aggregation Expressions

On this page

Aggregation expressions (page 471) can use both user-defined and system variables.

Variables can hold any *BSON type data* (page 186). To access the value of the variable, use a string with the variable name prefixed with double dollar signs (`$$`).

If the variable references an object, to access a specific field in the object, use the dot notation; i.e. `"$$<variable>.<field>"`.

[10] http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
[11] https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs
[12] https://www.mongodb.com/products/consulting?jmp=docs#database_modernization

User Variables

User variable names can contain the ascii characters `[_a-zA-Z0-9]` and any non-ascii character.

User variable names must begin with a lowercase ascii letter `[a-z]` or a non-ascii character.

System Variables

MongoDB offers the following system variables:

Variable	Description
ROOT	References the root document, i.e. the top-level document, currently being processed in the aggregation pipeline stage.
CURRENT	References the start of the field path being processed in the aggregation pipeline stage. Unless documented otherwise, all stages start with `CURRENT` (page 480) the same as `ROOT` (page 480). `CURRENT` (page 480) is modifiable. However, since `$<field>` is equivalent to `$$CURRENT.<field>`, rebinding `CURRENT` (page 480) changes the meaning of `$` accesses.
DESCEND	One of the allowed results of a `$redact` expression.
PRUNE	One of the allowed results of a `$redact` expression.
KEEP	One of the allowed results of a `$redact` expression.

See also:

`$let`, `$redact`, `$map`

CHAPTER 8

Indexes

8.1 Index Introduction

On this page

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a *collection scan*, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures [1] that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

The following diagram illustrates a query that selects and orders the matching documents using an index:

Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the *collection* level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

8.1.1 Index Types

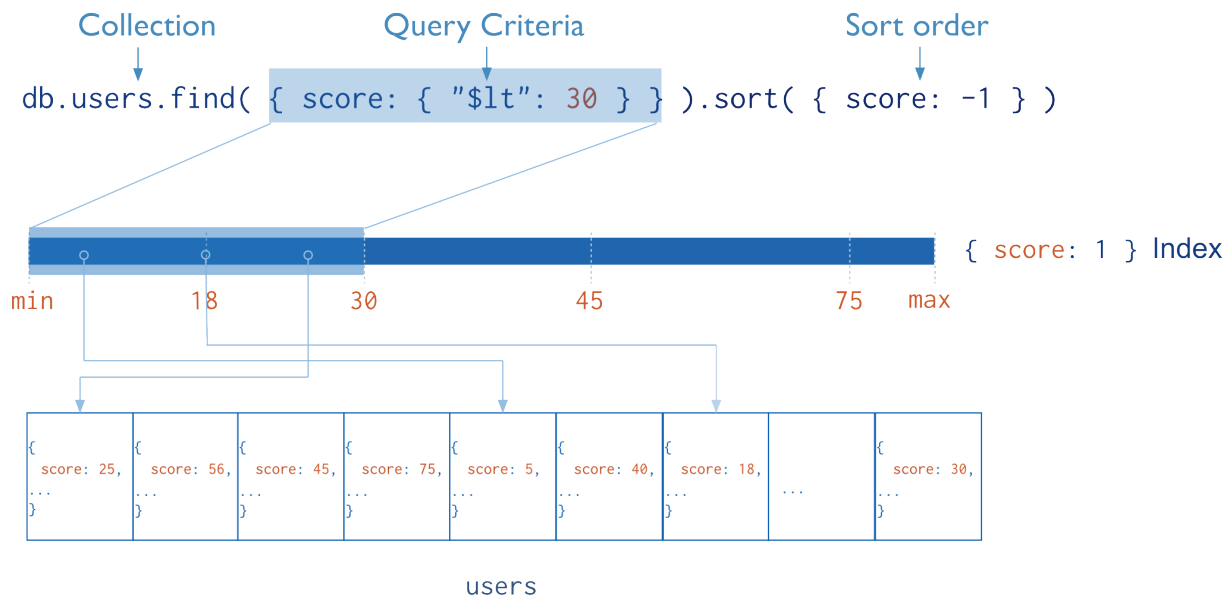
MongoDB provides a number of different index types to support specific types of data and queries.

Default `_id`

All MongoDB collections have an index on the `_id` field that exists by default. If applications do not specify a value for `_id` the driver or the `mongod` will create an `_id` field with an *ObjectId* value.

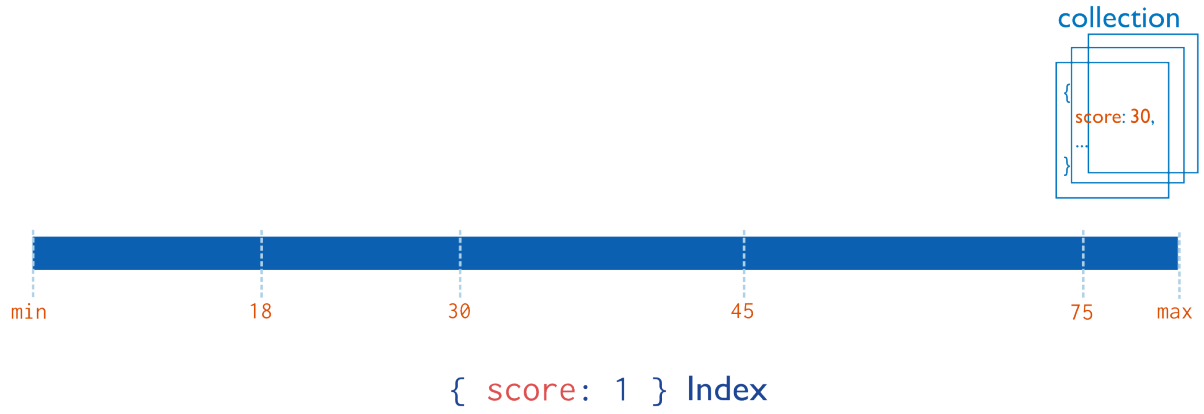
The `_id` index is *unique* and prevents clients from inserting two documents with the same value for the `_id` field.

[1] MongoDB indexes use a B-tree data structure.



Single Field

In addition to the MongoDB-defined `_id` index, MongoDB supports the creation of user-defined ascending/descending indexes on a *single field of a document* (page 487).



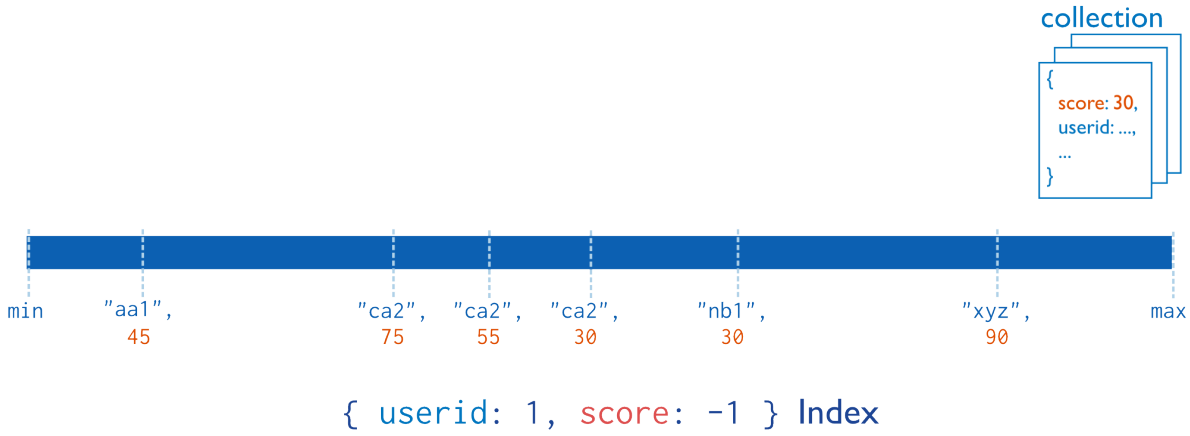
For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

See *Single Field Indexes* (page 487) and *Sort with a Single Field Index* (page 553) for more information on single-field indexes.

Compound Index

MongoDB also supports user-defined indexes on multiple fields, i.e. *compound indexes* (page 489).

The order of fields listed in a compound index has significance. For instance, if a compound index consists of `{ userid: 1, score: -1 }`, the index sorts first by `userid` and then, within each `userid` value, sorts by `score`.

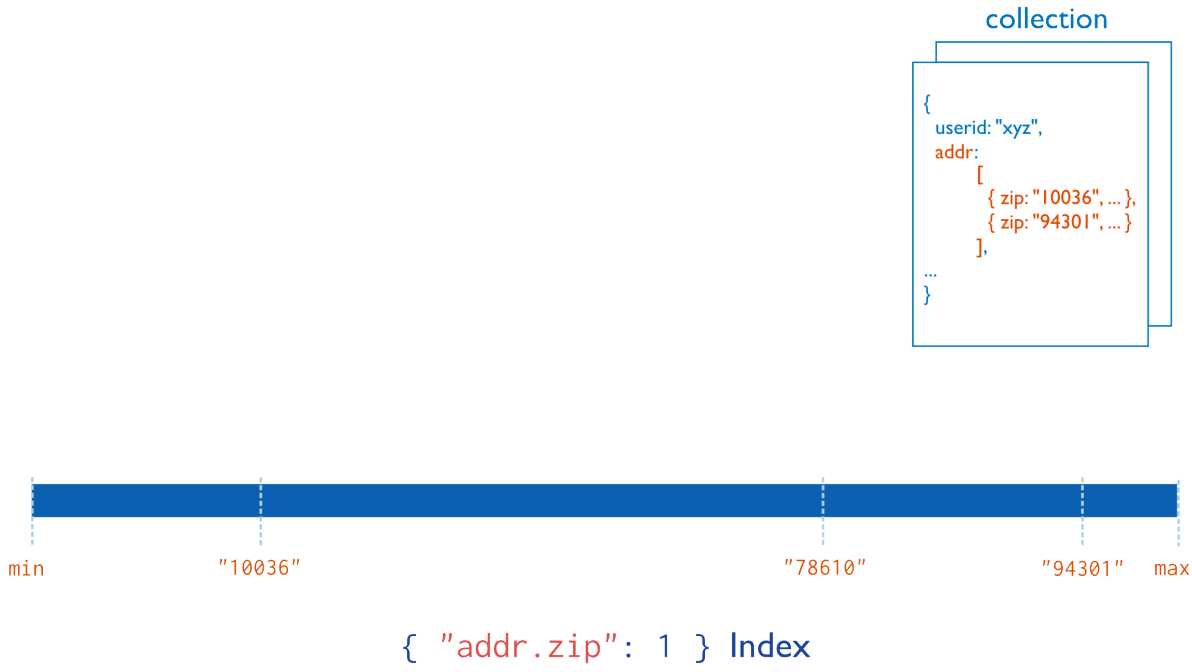


For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation. See *Sort Order* (page 490) for more information on the impact of index order on results in compound indexes.

See *Compound Indexes* (page 489) and *Sort on Multiple Fields* (page 553) for more information on compound indexes.

Multikey Index

MongoDB uses *multikey indexes* (page 491) to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for *every* element of the array. These *multikey indexes* (page 491) allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.



See *Multikey Indexes* (page 491) and *Multikey Index Bounds* (page 514) for more information on multikey indexes.

Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: *2d indexes* (page 498) that uses planar geometry when returning results and *2sphere indexes* (page 497) that use spherical geometry to return results.

See *2d Index Internals* (page 500) for a high level introduction to geospatial indexes.

Text Indexes

MongoDB provides a `text` index type that supports searching for string content in a collection. These text indexes do not store language-specific *stop* words (e.g. "the", "a", "or") and *stem* the words in a collection to only store root words.

See *Text Indexes* (page 501) for more information on text indexes and search.

Hashed Indexes

To support *hash based sharding* (page 689), MongoDB provides a *hashed index* (page 504) type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but *only* support equality matches and cannot support range-based queries.

8.1.2 Index Properties

Unique Indexes

The *unique* (page 506) property for an index causes MongoDB to reject duplicate values for the indexed field. To create a *unique index* (page 506) on a field that already has duplicate values, see *Drop Duplicates* (page 511) for index creation options. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Sparse Indexes

The *sparse* (page 507) property of an index ensures that the index only contain entries for documents that have the indexed field. The index skips documents that *do not* have the indexed field.

You can combine the sparse index option with the unique index option to reject documents that have duplicate values for a field but ignore documents that do not have the indexed key.

TTL Indexes

TTL indexes (page 504) are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

See: *Expire Data from Collections by Setting TTL* (page 222) for implementation instructions.

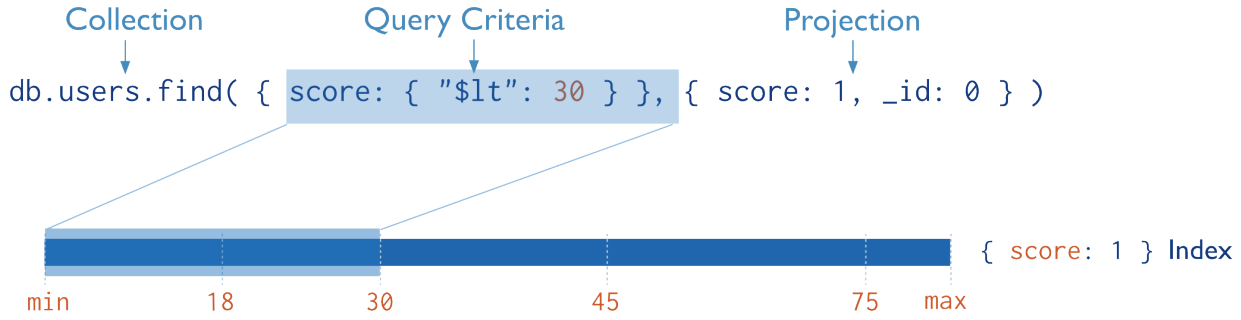
8.1.3 Index Use

Indexes can improve the efficiency of read operations. The *Analyze Query Performance* (page 117) tutorial provides an example of the execution statistics of a query with and without an index.

For information on how MongoDB chooses an index to use, see *query optimizer* (page 72).

8.1.4 Covered Queries

When the query criteria and the *projection* of a query include *only* the indexed fields, MongoDB will return results directly from the index *without* scanning any documents or bringing documents into memory. These covered queries can be *very* efficient.



For more information on covered queries, see *Covered Query* (page 71).

8.1.5 Index Intersection

New in version 2.6.

MongoDB can use the *intersection of indexes* (page 512) to fulfill queries. For queries that specify compound query conditions, if one index can fulfill a part of a query condition, and another index can fulfill another part of the query condition, then MongoDB can use the intersection of the two indexes to fulfill the query. Whether the use of a compound index or the use of an index intersection is more efficient depends on the particular query and the system.

For details on index intersection, see *Index Intersection* (page 512).

8.1.6 Restrictions

Certain restrictions apply to indexes, such as the length of the index keys or the number of indexes per collection. See *Index Limitations* for details.

8.2 Index Concepts

These documents describe and provide examples of the types, configuration options, and behavior of indexes in MongoDB. For an over view of indexing, see *Index Introduction* (page 481). For operational instructions, see *Indexing Tutorials* (page 519). The *Indexing Reference* (page 556) documents the commands and operations specific to index construction, maintenance, and querying in MongoDB, including index types and creation options.

8.2.1 Index Types

MongoDB provides a number of different index types. You can create indexes on any field or embedded field within a document or embedded document.

In general, you should create indexes that support your common and user-facing queries. Having these indexes will ensure that MongoDB scans the smallest possible number of documents.

In the `mongo` shell, you can create an index by calling the `ensureIndex()` method. For more detailed instructions about building indexes, see the *Indexing Tutorials* (page 519) page.

Single Field Indexes **(page 487)** A single field index only includes data from a single field of the documents in a collection. MongoDB supports single field indexes on fields at the top level of a document *and* on fields in sub-documents.

Compound Indexes **(page 489)** A compound index includes more than one field of the documents in a collection.

Multikey Indexes **(page 491)** A multikey index is an index on an array field, adding an index key for each value in the array.

Geospatial Indexes and Queries **(page 494)** Geospatial indexes support location-based searches on data that is stored as either GeoJSON objects or legacy coordinate pairs.

Text Indexes **(page 501)** Text indexes support search of string content in documents.

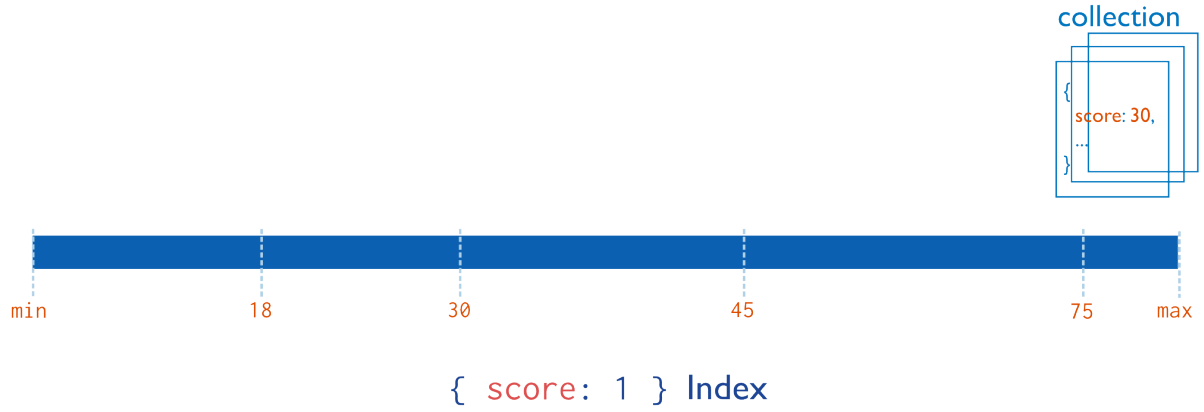
Hashed Index **(page 504)** Hashed indexes maintain entries with hashes of the values of the indexed field and are primarily used with sharded clusters to support hashed shard keys.

Single Field Indexes

On this page

MongoDB provides complete support for indexes on any field in a *collection* of *documents*. By default, all collections have an index on the *_id field* (page 487), and applications and users may add additional indexes to support important queries and operations.

MongoDB supports indexes that contain either a single field *or* multiple fields depending on the operations that this index-type supports. This document describes ascending/descending indexes that contain a single field. Consider the following illustration of a single field index.



See also:

Compound Indexes (page 489) for information about indexes that include multiple fields, and *Index Introduction* (page 481) for a higher level introduction to indexing in MongoDB.

Example

Given the following document in the `friends` collection:

```
{ "_id" : ObjectId(...),
  "name" : "Alice",
  "age" : 27
}
```

The following command creates an index on the `name` field:

```
db.friends.ensureIndex( { "name" : 1 } )
```

Cases

`_id` Field Index MongoDB creates the `_id` index, which is an ascending *unique index* (page 506) on the `_id` field, for all collections when the collection is created. You cannot remove the index on the `_id` field.

Think of the _id field as the *primary key* for a collection. Every document *must* have a unique _id field. You may store any unique value in the _id field. The default value of _id is an *ObjectId* which is generated when the client inserts the document. An *ObjectId* is a 12-byte unique identifier suitable for use as the value of an _id field.

Note: In *sharded clusters*, if you do *not* use the _id field as the *shard key*, then your application **must** ensure the uniqueness of the values in the _id field to prevent errors. This is most-often done by using a standard auto-generated *ObjectId*.

Before version 2.2, *capped collections* did not have an _id field. In version 2.2 and newer, capped collections do have an _id field, except those in the local *database*. See *Capped Collections Recommendations and Restrictions* (page 220) for more information.

Indexes on Embedded Fields You can create indexes on fields within embedded documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from *indexes on embedded documents* (page 488), which include the full content up to the maximum index size of the embedded document in the index. Instead, indexes on embedded fields allow you to use a "dot notation," to introspect into embedded documents.

Consider a collection named people that holds documents that resemble the following example document:

```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

You can create an index on the address.zipcode field, using the following specification:

```
db.people.ensureIndex( { "address.zipcode": 1 } )
```

Indexes on Embedded Documents You can also create indexes on embedded documents.

For example, the factories collection contains documents that contain a metro field, such as:

```
{
  _id: ObjectId(...),
  metro: {
            city: "New York",
            state: "NY"
          },
  name: "Giant Factory"
}
```

The metro field is an embedded document, containing the embedded fields city and state. The following command creates an index on the metro field as a whole:

```
db.factories.ensureIndex( { metro: 1 } )
```

The following query can use the index on the metro field:

```
db.factories.find( { metro: { city: "New York", state: "NY" } } )
```

This query returns the above document. When performing equality matches on embedded documents, field order matters and the embedded documents must match exactly. For example, the following query does not match the above document:

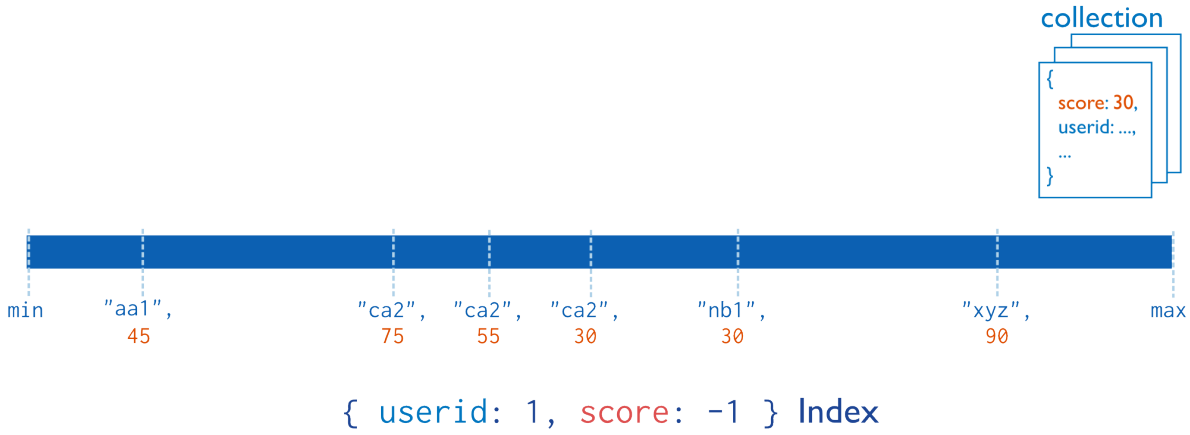
```
db.factories.find( { metro: { state: "NY", city: "New York" } } )
```

See *query-embedded-documents* for more information regarding querying on embedded documents.

Compound Indexes

On this page

MongoDB supports *compound indexes*, where a single index structure holds references to multiple fields [2] within a collection's documents. The following diagram illustrates an example of a compound index on two fields:



Compound indexes can support queries that match on multiple fields.

Example

Consider a collection named `products` that holds documents that resemble the following document:

```
{
 "_id": ObjectId(...),
 "item": "Banana",
 "category": ["food", "produce", "grocery"],
 "location": "4th Street Store",
 "stock": 4,
 "type": "cases",
 "arrival": Date(...)
}
```

If applications query on the `item` field as well as query on both the `item` field and the `stock` field, you can specify a single compound index to support both of these queries:

```
db.products.ensureIndex( { "item": 1, "stock": 1 } )
```

[2] MongoDB imposes a `limit of 31 fields for any compound index`.

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 504).

The order of the fields in a compound index is very important. In the previous example, the index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by values of the `stock` field. See *Sort Order* (page 490) for more information.

In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index fields. For details, see *Prefixes* (page 490).

Sort Order

Indexes store references to fields in either ascending (`1`) or descending (`-1`) sort order. For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for *compound indexes* (page 489), sort order can matter in determining whether the index can support a sort operation.

Consider a collection `events` that contains documents with the fields `username` and `date`. Applications can issue queries that return results sorted first by ascending `username` values and then by descending (i.e. more recent to last) `date` values, such as:

```
db.events.find().sort( { username: 1, date: -1 } )
```

or queries that return results sorted first by descending `username` values and then by ascending `date` values, such as:

```
db.events.find().sort( { username: -1, date: 1 } )
```

The following index can support both these sort operations:

```
db.events.ensureIndex( { "username" : 1, "date" : -1 } )
```

However, the above index **cannot** support sorting by ascending `username` values and then by ascending `date` values, such as the following:

```
db.events.find().sort( { username: 1, date: 1 } )
```

For more information on sort order and compound indexes, see *Use Indexes to Sort Query Results* (page 553).

Prefixes

Index prefixes are the *beginning* subsets of indexed fields. For example, consider the following compound index:

```
{ "item": 1, "location": 1, "stock": 1 }
```

The index has the following index prefixes:

- `{ item: 1 }`
- `{ item: 1, location: 1 }`

For a compound index, MongoDB can use the index to support queries on the index prefixes. As such, MongoDB can use the index for queries on the following fields:

- the `item` field,
- the `item` field *and* the `location` field,
- the `item` field *and* the `location` field *and* the `stock` field.

MongoDB can also use the index to support a query on `item` and `stock` fields since `item` field corresponds to a prefix. However, the index would not be as efficient in supporting the query as would be an index on only `item` and `stock`.

However, MongoDB cannot use the index to support queries that include the following fields since without the `item` field, none of the listed fields correspond to a prefix index:

- the `location` field,
- the `stock` field, or
- the `location` and `stock` fields.

If you have a collection that has both a compound index and an index on its prefix (e.g. `{ a: 1, b: 1 }` and `{ a: 1 }`), if neither index has a sparse or unique constraint, then you can remove the index on the prefix (e.g. `{ a: 1 }`). MongoDB will use the compound index in all of the situations that it would have used the prefix index.

Index Intersection

Starting in version 2.6, MongoDB can use *index intersection* (page 512) to fulfill queries. The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system. See *Index Intersection and Compound Indexes* (page 513) for more details.

Multikey Indexes

On this page

To index a field that holds an array value, MongoDB creates an index key for each element in the array. These *multikey* indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values (e.g. strings, numbers) *and* nested documents.

Create Multikey Index

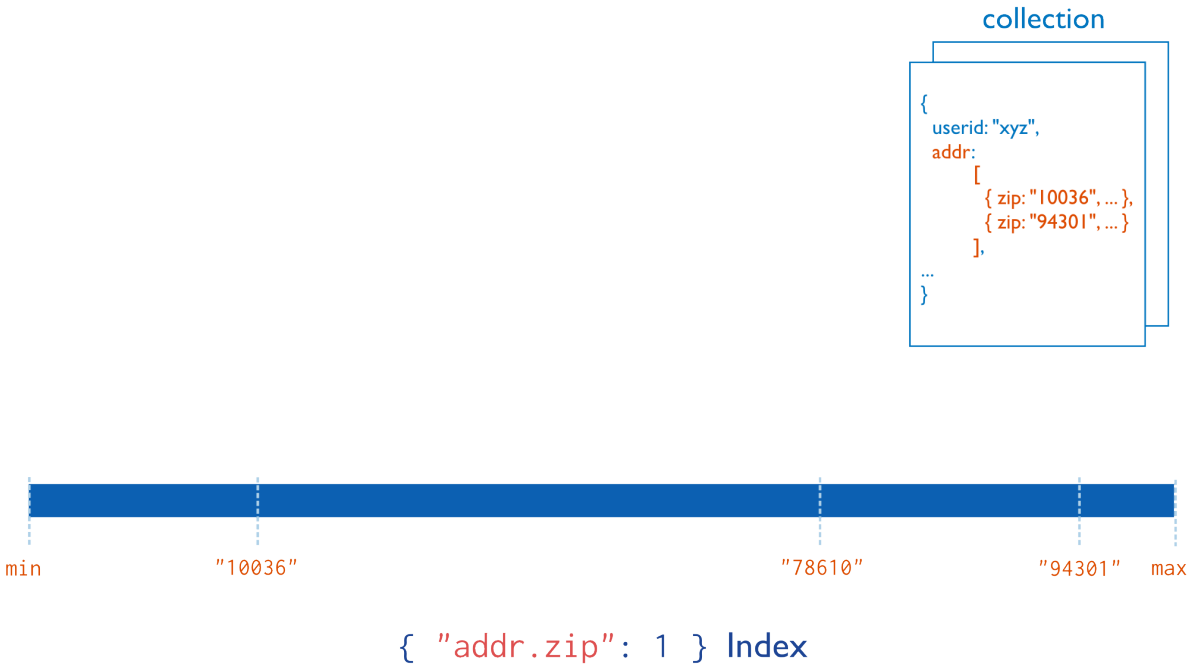
To create a multikey index, use the `db.collection.createIndex()` method:

```
db.coll.createIndex( { <field>: < 1 or -1 > } )
```

MongoDB automatically creates a multikey index if any indexed field is an array; you do not need to explicitly specify the multikey type.

Index Bounds

If an index is multikey, then computation of the index bounds follows special rules. For details on multikey index bounds, see *Multikey Index Bounds* (page 514).



Limitations

Compound Multikey Indexes For a *compound* (page 489) multikey index, each indexed document can have *at most* one indexed field whose value is an array. As such, you cannot create a compound multikey index if more than one to-be-indexed field of a document is an array. Or, if a compound multikey index already exists, you cannot insert a document that would violate this restriction.

For example, consider a collection that contains the following document:

```
{ _id: 1, a: [ 1, 2 ], b: [ 1, 2 ], category: "AB - both arrays" }
```

You cannot create a compound multikey index `{ a: 1, b: 1 }` on the collection since both the `a` and `b` fields are arrays.

But consider a collection that contains the following documents:

```
{ _id: 1, a: [1, 2], b: 1, category: "A array" }
{ _id: 2, a: 1, b: [1, 2], category: "B array" }
```

A compound multikey index `{ a: 1, b: 1 }` is permissible since for each document, only one field indexed by the compound multikey index is an array; i.e. no document contains array values for both `a` and `b` fields. After creating the compound multikey index, if you attempt to insert a document where both `a` and `b` fields are arrays, MongoDB will fail the insert.

Shard Keys You **cannot** specify a multikey index as the shard key index.

Changed in version 2.6: However, if the shard key index is a *prefix* (page 490) of a compound index, the compound index is allowed to become a compound *multikey* index if one of the other keys (i.e. keys that are not part of the shard key) indexes an array. Compound multikey indexes can have an impact on performance.

Hashed Indexes *Hashed* (page 504) indexes **cannot** be multikey.

Covered Queries A *multikey index* (page 491) cannot support a *covered query* (page 71).

Examples

Index Basic Arrays Consider a `survey` collection with the following document:

```
{ _id: 1, item: "ABC", ratings: [ 2, 5, 9 ] }
```

Create an index on the field `ratings`:

```
db.survey.createIndex( { ratings: 1 } )
```

Since the `ratings` field contains an array, the index on `ratings` is multikey. The multikey index contains the following three index keys, each pointing to the same document:

- 2,
- 5, and
- 9.

Index Arrays with Embedded Documents You can create multikey indexes on array fields that contain nested objects.

Consider an `inventory` collection with documents of the following form:

```
{
  _id: 1,
  item: "abc",
  stock: [
    { size: "S", color: "red", quantity: 25 },
    { size: "S", color: "blue", quantity: 10 },
    { size: "M", color: "blue", quantity: 50 }
  ]
}
{
  _id: 2,
  item: "def",
  stock: [
    { size: "S", color: "blue", quantity: 20 },
    { size: "M", color: "blue", quantity: 5 },
    { size: "M", color: "black", quantity: 10 },
    { size: "L", color: "red", quantity: 2 }
  ]
}
{
  _id: 3,
  item: "ijk",
  stock: [
    { size: "M", color: "blue", quantity: 15 },
    { size: "L", color: "blue", quantity: 100 },
    { size: "L", color: "red", quantity: 25 }
  ]
}

...
```

The following operation creates a multikey index on the `stock.size` and `stock.quantity` fields:

```
db.inventory.createIndex( { "stock.size": 1, "stock.quantity": 1 } )
```

The compound multikey index can support queries with predicates that include both indexed fields as well as predicates that include only the index prefix `"stock.size"`, as in the following examples:

```
db.inventory.find( { "stock.size": "M" } )
db.inventory.find( { "stock.size": "S", "stock.quantity": { $gt: 20 } } )
```

For details on how MongoDB can combine multikey index bounds, see *Multikey Index Bounds* (page 514). For more information on behavior of compound indexes and prefixes, see *compound indexes and prefixes* (page 490).

The compound multikey index can also support sort operations, such as the following examples:

```
db.inventory.find( ).sort( { "stock.size": 1, "stock.quantity": 1 } )
db.inventory.find( { "stock.size": "M" } ).sort( { "stock.quantity": 1 } )
```

For more information on behavior of compound indexes and sort operations, see *Use Indexes to Sort Query Results* (page 553).

Geospatial Indexes and Queries

On this page

MongoDB offers a number of indexes and query mechanisms to handle geospatial information. This section introduces MongoDB's geospatial features. For complete examples of geospatial queries in MongoDB, see *Geospatial Index Tutorials* (page 533).

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 497) 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 498) index.

Location Data

If you choose spherical surface calculations, you store location data as either:

GeoJSON Objects Queries on *GeoJSON* objects always calculate on a sphere. The default coordinate reference system for GeoJSON uses the *WGS84* datum.

New in version 2.4: Support for GeoJSON storage and queries is new in version 2.4. Prior to version 2.4, all geospatial data used coordinate pairs.

Changed in version 2.6: Support for additional GeoJSON types: MultiPoint, MultiLineString, MultiPolygon, GeometryCollection.

MongoDB supports the following GeoJSON objects:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection

Legacy Coordinate Pairs MongoDB supports spherical surface calculations on *legacy coordinate pairs* using a `2dsphere` index by converting the data to the GeoJSON Point type.

If you choose flat surface calculations via a `2d` index, you can store data only as *legacy coordinate pairs*.

Query Operations

MongoDB's geospatial query operators let you query for:

Inclusion MongoDB can query for locations contained entirely within a specified polygon. Inclusion queries use the `$geoWithin` operator.

Both `2d` and `2dsphere` indexes can support inclusion queries. MongoDB does not require an index for inclusion queries; however, such indexes will improve query performance.

Intersection MongoDB can query for locations that intersect with a specified geometry. These queries apply only to data on a spherical surface. These queries use the `$geoIntersects` operator.

Only `2dsphere` indexes support intersection.

Proximity MongoDB can query for the points nearest to another point. Proximity queries use the `$near` operator. The `$near` operator requires a `2d` or `2dsphere` index.

Geospatial Indexes

MongoDB provides the following geospatial index types to support the geospatial queries.

2dsphere *2dsphere* (page 497) indexes support:

- Calculations on a sphere
- GeoJSON objects and include backwards compatibility for legacy coordinate pairs
- Compound indexes with scalar index fields (i.e. ascending or descending) as a prefix or suffix of the `2dsphere` index field

New in version 2.4: `2dsphere` indexes are not available before version 2.4.

See also:

Query a 2dsphere Index (page 535)

2d *2d* (page 498) indexes support:

- Calculations using flat geometry
- Legacy coordinate pairs (i.e., geospatial points on a flat coordinate system)
- Compound indexes with only one additional field, as a suffix of the `2d` index field

See also:

Query a 2d Index (page 538)

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 it uses fields other than the shard key fields.

For sharded collections, queries using `$near` and `$nearSphere` are not supported. You can instead use either the `geoNear` command or the `$geoNear` aggregation stage.

You can also query for geospatial data using `$geoWithin`.

Additional Resources

The following pages provide complete documentation for geospatial indexes and queries:

***2dsphere Indexes* (page 497)** A `2dsphere` index supports queries that calculate geometries on an earth-like sphere. The index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

***2d Indexes* (page 498)** The `2d` index supports data stored as legacy coordinate pairs and is intended for use in MongoDB 2.2 and earlier.

***geoHaystack Indexes* (page 499)** A haystack index is a special index optimized to return results over small areas. For queries that use spherical geometry, a `2dsphere` index is a better option than a haystack index.

***2d Index Internals* (page 500)** Provides a more in-depth explanation of the internals of geospatial indexes. This material is not necessary for normal operations but may be useful for troubleshooting and for further understanding.

2dsphere Indexes

On this page

New in version 2.4.

Overview A 2dsphere index supports queries that calculate geometries on an earth-like sphere. 2dsphere index supports all MongoDB geospatial queries: queries for inclusion, intersection and proximity. See the http://docs.mongodb.org/manual/reference/operator/query-geospatial for the query operators that support geospatial queries.

The 2dsphere index supports data stored as *GeoJSON* (page 558) objects and as legacy coordinate pairs (See also *2dsphere Indexed Field Restrictions* (page 498)). For legacy coordinate pairs, the index converts the data to GeoJSON *Point* (page 558). For details on the supported GeoJSON objects, see *GeoJSON Objects* (page 558).

The default datum for an earth-like sphere is *WGS84*. Coordinate-axis order is **longitude, latitude**.

2dsphere (Version 2) Changed in version 2.6.

MongoDB 2.6 introduces a version 2 of 2dsphere indexes. Version 2 is the default version of 2dsphere indexes created in MongoDB 2.6 and later series. To override the default version 2 and create a version 1 index, include the option { "2dsphereIndexVersion": 1 } when creating the index.

sparse Property Changed in version 2.6.

2dsphere (Version 2) indexes are *sparse* (page 507) by default and ignores the *sparse: true* (page 507) option. If a document lacks a 2dsphere index field (or the field is null or an empty array), MongoDB does not add an entry for the document to the index. For inserts, MongoDB inserts the document but does not add to the 2dsphere index.

For a compound index that includes a 2dsphere index key along with keys of other types, only the 2dsphere index field determines whether the index references a document.

Earlier versions of MongoDB only support 2dsphere (Version 1) indexes. 2dsphere (Version 1) indexes are *not* sparse by default and will reject documents with null location fields.

Additional GeoJSON Objects 2dsphere (Version 2) includes support for additional GeoJSON object: *MultiPoint* (page 560), *MultiLineString* (page 560), *MultiPolygon* (page 561), and *GeometryCollection* (page 561). For details on all supported GeoJSON objects, see *GeoJSON Objects* (page 558).

Considerations

geoNear and $geoNear Restrictions The geoNear command and the $geoNear pipeline stage require that a collection have *at most* only one 2dsphere index and/or only one *2d* (page 498) index whereas *geospatial query operators* (e.g. $near and $geoWithin) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the geoNear command and the $geoNear pipeline stage exists because neither the geoNear command nor the $geoNear pipeline stage syntax includes the location field. As such, index selection among multiple 2d indexes or 2dsphere indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

Shard Key Restrictions 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.

`2dsphere` Indexed Field Restrictions Fields with *2dsphere* (page 497) indexes must hold geometry data in the form of *coordinate pairs* or *GeoJSON* data. If you attempt to insert a document with non-geometry data in a `2dsphere` indexed field, or build a `2dsphere` index on a collection where the indexed field has non-geometry data, the operation will fail.

Create a `2dsphere` Index To create a `2dsphere` index, use the `db.collection.ensureIndex()` method, specifying the location field as the key and specify the string literal `"2dsphere"` as the index type:

```
db.collection.ensureIndex( { <location field> : "2dsphere" } )
```

Unlike a compound *2d* (page 498) index which can reference one location field and one other field, a *compound* (page 489) `2dsphere` index can reference multiple location and non-location fields.

For more information on creating `2dspshere` indexes, see *Create a 2dsphere Index* (page 533).

`2d` Indexes

On this page

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.

See the `http://docs.mongodb.org/manual/reference/operator/query-geospatial` for the query operators that support geospatial queries.

Considerations The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2d` index and/or only one *2dsphere index* (page 497) whereas *geospatial query operators* (e.g. `$near` and `$geoWithin`) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the `geoNear` command and the `$geoNear` pipeline stage exists because neither the `geoNear` command nor the `$geoNear` pipeline stage syntax includes the location field. As such, index selection among multiple `2d` indexes or `2dsphere` indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

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 497) index.

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.

Behavior The `2d` index supports calculations on a flat, Euclidean plane. The `2d` index also supports *distance-only* calculations on a sphere, but for *geometric* calculations (e.g. `$geoWithin`) on a sphere, store data as GeoJSON objects and use the `2dsphere` index type.

A `2d` index can reference two fields. The first must be the location field. A `2d` compound index constructs queries that select first on the location field, and then filters those results by the additional criteria. A compound `2d` index can cover queries.

Points on a 2D Plane To store location data as legacy coordinate pairs, use an array or an embedded document. When possible, use the array format:

```
loc : [ <longitude> , <latitude> ]
```

Consider the embedded document form:

```
loc : { lng : <longitude> , lat : <latitude> }
```

Arrays are preferred as certain languages do not guarantee associative map ordering.

For all points, if you use longitude and latitude, store coordinates in **longitude, latitude** order.

`sparse` Property `2d` indexes are *sparse* (page 507) by default and ignores the *sparse: true* (page 507) option. If a document lacks a `2d` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `2d` index. For inserts, MongoDB inserts the document but does not add to the `2d` index.

For a compound index that includes a `2d` index key along with keys of other types, only the `2d` index field determines whether the index references a document.

`geoHaystack` Indexes

On this page

A `geoHaystack` index is a special index that is optimized to return results over small areas. `geoHaystack` indexes improve performance on queries that use flat geometry.

For queries that use spherical geometry, a **2dsphere index is a better option** than a haystack index. *2dsphere indexes* (page 497) allow field reordering; `geoHaystack` indexes require the first field to be the location field. Also, `geoHaystack` indexes are only usable via commands and so always return all results at once.

Behavior `geoHaystack` 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 `geoHaystack` index contains all the documents within a specified proximity to a given longitude and latitude.

`sparse` Property `geoHaystack` indexes are *sparse* (page 507) by default and ignore the *sparse: true* (page 507) option. If a document lacks a `geoHaystack` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `geoHaystack` index. For inserts, MongoDB inserts the document but does not add to the `geoHaystack` index.

`geoHaystack` indexes include one `geoHaystack` index key and one non-geospatial index key; however, only the `geoHaystack` index field determines whether the index references a document.

Create `geoHaystack` Index To create a `geoHaystack` index, see *Create a Haystack Index* (page 540). For information and example on querying a haystack index, see *Query a Haystack Index* (page 540).

2d Index Internals

On this page

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 537) and then indexes the geohash values.

To calculate a geohash value, recursively divide a two-dimensional map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

```
01  11

00  10
```

These two-bit values (`00`, `01`, `10`, and `11`) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of `00`. The top left quadrant would have the geohash of `01`. The bottom right and top right would have a geohash of `10` and `11`, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is `11`, and the geohash for the sub-quadrants would be (clockwise from the top left): `1101`, `1111`, `1110`, and `1100`, respectively.

Multi-location Documents for `2d` Indexes New in version 2.0: Support for multiple locations in a document.

While `2d` geospatial indexes do not support more than one set of coordinates in a document, you can use a *multi-key index* (page 491) 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 an embedded 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" } )
```

To include the location field with the distance field in multi-location document queries, specify `includeLocs: true` in the `geoNear` command.

Text Indexes

On this page

New in version 2.4.

MongoDB provides `text` indexes to support text search of string content in documents of a collection.

`text` indexes can include any field whose value is a string or an array of string elements. To perform queries that access the `text` index, use the `$text` query operator.

Changed in version 2.6: MongoDB enables the text search feature by default. In MongoDB 2.4, you need to enable the text search feature manually to create `text` indexes and perform *text search* (page 503).

Create Text Index

To create a `text` index, use the `db.collection.ensureIndex()` method. To index a field that contains a string or an array of string elements, include the field and specify the string literal `"text"` in the index document, as in the following example:

```
db.reviews.ensureIndex( { comments: "text" } )
```

A collection can have at most **one** `text` index.

However, you can specify multiple fields for the `text` index. For examples of creating `text` indexes on multiple fields, see *Create a text Index* (page 543) and *Wildcard Text Indexes* (page 502).

Wildcard Text Indexes

To allow for text search on all fields with string content, use the wildcard specifier (`$**`) to index all fields in the collection that contain string content. Such an index can be useful with highly unstructured data if it is unclear which fields to include in the text index or for ad-hoc querying.

With a wildcard text index, MongoDB indexes every field that contains string data for each document in the collection. The following example creates a text index using the wildcard specifier:

```
db.collection.createIndex( { "$**": "text" } )
```

Wildcard text indexes are `text` indexes on multiple fields. As such, you can assign weights to specific fields during index creation to control the ranking of the results. For more information using weights to control the results of a text search, see *Control Search Results with Weights* (page 547).

Wildcard text indexes, as with all text indexes, can be part of a compound indexes. For example, the following creates a compound index on the field `a` as well as the wildcard specifier:

```
db.collection.createIndex( { a: 1, "$**": "text" } )
```

As with all *compound text indexes* (page 503), since the `a` precedes the text index key, in order to perform a `$text` search with this index, the query predicate must include an equality match conditions `a`. For information on compound text indexes, see *Compound Text Indexes* (page 503).

Supported Languages and Stop Words

MongoDB supports text search for various languages. `text` indexes drop language-specific stop words (e.g. in English, "the", "an", "a", "and", etc.) and uses simple language-specific suffix stemming. For a list of the supported languages, see *Text Search Languages* (page 561).

If you specify a language value of `"none"`, then the `text` index uses simple tokenization with no list of stop words and no stemming.

For the Latin alphabet, `text` indexes are case insensitive for non-diacritics; i.e. case insensitive for `[A-z]`. For all other characters, text indexes treat them as distinct.

To specify a language for the `text` index, see *Specify a Language for Text Index* (page 544).

`sparse` Property

`text` indexes are *sparse* (page 507) by default and ignores the *sparse: true* (page 507) option. If a document lacks a `text` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `text` index. For inserts, MongoDB inserts the document but does not add to the `text` index.

For a compound index that includes a `text` index key along with keys of other types, only the `text` index field determine whether the index references a document. The other keys do not determine whether the index references the documents or not.

Restrictions

Text Search and Hints You cannot use `hint()` if the query includes a `$text` query expression.

Text Index and Sort Sort operations cannot obtain sort order from a `text` index, even from a *compound text index* (page 503); i.e. sort operations cannot use the ordering in the text index.

Compound Index A *compound index* (page 489) can include a `text` index key in combination with ascending/descending index keys. However, these compound indexes have the following restrictions:

- A compound `text` index cannot include any other special index types, such as *multi-key* (page 491) or *geospatial* (page 495) index fields.
- If the compound `text` index includes keys **preceding** the `text` index key, to perform a `$text` search, the query predicate must include **equality match conditions** on the preceding keys.

See also *Text Index and Sort* (page 503) for additional limitations.

For an example of a compound text index, see *Limit the Number of Entries Scanned* (page 548).

Drop a Text Index To drop a `text` index, pass the name of the index to the `db.collection.dropIndex()` method. To get the name of the index, run the `getIndexes()` method.

For information on the default naming scheme for `text` indexes as well as overriding the default name, see *Specify Name for text Index* (page 546).

Storage Requirements and Performance Costs

`text` indexes have the following storage requirements and performance costs:

- `text` indexes change the space allocation method for all future record allocations in a collection to `usePowerOf2Sizes`.
- `text` indexes can be large. They contain one index entry for each unique post-stemmed word in each indexed field for each document inserted.
- Building a `text` index is very similar to building a large multi-key index and will take longer than building a simple ordered (scalar) index on the same data.
- When building a large `text` index on an existing collection, ensure that you have a sufficiently high limit on open file descriptors. See the *recommended settings* (page 300).
- `text` indexes will impact insertion throughput because MongoDB must add an index entry for each unique post-stemmed word in each indexed field of each new source document.
- Additionally, `text` indexes do not store phrases or information about the proximity of words in the documents. As a result, phrase queries will run much more effectively when the entire collection fits in RAM.

Text Search

Text search supports the search of string content in documents of a collection. MongoDB provides the `$text` operator to perform text search in queries and in *aggregation pipelines* (page 549).

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.

The `$text` operator can search for words and phrases. The query 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 the `$text` query operator. For examples of text search in aggregation pipeline, see *Text Search in the Aggregation Pipeline* (page 549).

Hashed Index

New in version 2.4.

Hashed indexes maintain entries with hashes of the values of the indexed field. The hashing function collapses embedded documents and computes the hash for the entire value but does not support multi-key (i.e. arrays) indexes.

Hashed indexes support *sharding* (page 675) a collection using a *hashed shard key* (page 689). Using a hashed shard key to shard a collection ensures a more even distribution of data. See *Shard a Collection Using a Hashed Shard Key* (page 711) for more details.

MongoDB can use the `hashed` index to support equality queries, but `hashed` indexes do not support range queries.

You may not create compound indexes that have `hashed` index fields or specify a unique constraint on a `hashed` index; however, you can create both a `hashed` index and an ascending/descending (i.e. non-hashed) index on the same field: MongoDB will use the scalar index for range queries.

Warning: MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be reliably converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

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.

8.2.2 Index Properties

In addition to the numerous *index types* (page 486) MongoDB supports, indexes can also have various properties. The following documents detail the index properties that you can select when building an index.

***TTL Indexes* (page 504)** The TTL index is used for TTL collections, which expire data after a period of time.

***Unique Indexes* (page 506)** A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

***Sparse Indexes* (page 507)** A sparse index does not index documents that do not have the indexed field.

TTL Indexes

On this page

TTL indexes are special single-field indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. Data expiration is useful for certain types of information like machine generated event data, logs, and session information that only need to persist in a database for a finite amount of time.

To create a TTL index, use the `db.collection.ensureIndex()` method with the `expireAfterSeconds` option on a field whose value is either a *date* (page 189) or an array that contains *date values* (page 189).

For example, to create a TTL index on the `lastModifiedDate` field of the `eventlog` collection, use the following operation in the `mongo` shell:

```
db.eventlog.ensureIndex( { "lastModifiedDate": 1 }, { expireAfterSeconds: 3600 } )
```

Behavior

Expiration of Data TTL indexes expire documents after the specified number of seconds has passed since the indexed field value; i.e. the expiration threshold is the indexed field value plus the specified number of seconds.

If the field is an array, and there are multiple date values in the index, MongoDB uses *lowest* (i.e. earliest) date value in the array to calculate the expiration threshold.

If the indexed field in a document is not a *date* or an array that holds a date value(s), the document will not expire.

If a document does not contain the indexed field, the document will not expire.

Delete Operations A background thread in `mongod` reads the values in the index and removes expired *documents* from the collection.

When the TTL thread is active, you will see *delete* (page 77) operations in the output of `db.currentOp()` or in the data collected by the *database profiler* (page 239).

Timing of the Delete Operation When you build a TTL index in the *background* (page 510), the TTL thread can begin deleting documents while the index is building. If you build a TTL index in the foreground, MongoDB begins removing expired documents as soon as the index finishes building.

The TTL index does not guarantee that expired data will be deleted immediately upon expiration. There may be a delay between the time a document expires and the time that MongoDB removes the document from the database.

The background task that removes expired documents runs *every 60 seconds*. As a result, documents may remain in a collection during the period between the expiration of the document and the running of the background task.

Because the duration of the removal operation depends on the workload of your `mongod` instance, expired data may exist for some time *beyond* the 60 second period between runs of the background task.

Replica Sets On *replica sets*, the TTL background thread *only* deletes documents on the *primary*. However, the TTL background thread does run on secondaries. *Secondary* members replicate deletion operations from the primary.

Support for Queries A TTL index supports queries in the same way non-TTL indexes do.

Record Allocation A collection with a TTL index has `usePowerOf2Sizes` enabled, and you cannot modify this setting for the collection. As a result of enabling `usePowerOf2Sizes`, MongoDB must allocate more disk space relative to data size. This approach helps mitigate the possibility of storage fragmentation caused by frequent delete operations and leads to more predictable storage use patterns.

Restrictions

- TTL indexes are a single-field indexes. *Compound indexes* (page 489) do not support TTL and ignores the `expireAfterSeconds` option.
- The `_id` field does not support TTL indexes.
- You cannot create a TTL index on a *capped collection* (page 219) because MongoDB cannot remove documents from a capped collection.
- You cannot use `ensureIndex()` to change the value of `expireAfterSeconds` of an existing index. Instead use the `collMod` database command in conjunction with the `index` collection flag. Otherwise, to change the value of the option of an existing index, you must drop the index first and recreate.
- If a non-TTL single-field index already exists for a field, you cannot create a TTL index on the same field since you cannot create indexes that have the same key specification and differ only by the options. To change a non-TTL single-field index to a TTL index, you must drop the index first and recreate with the `expireAfterSeconds` option.

Additional Information

For examples, see *Expire Data from Collections by Setting TTL* (page 222).

Unique Indexes

On this page

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

To create a unique index, use the `db.collection.ensureIndex()` method with the `unique` option set to `true`. For example, to create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` shell:

```
db.members.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 489), then MongoDB will enforce uniqueness on the *combination* of values rather than the individual value for any or all values of the key.

Behavior

Unique Constraint Across Separate Documents The unique constraint applies to separate documents in the collection. That is, the unique index prevents *separate* documents from having the same value for the indexed key, but the index does not prevent a document from having multiple elements or embedded documents in an indexed array from

having the same value. In the case of a single document with repeating values, the repeated value is inserted into the index only once.

For example, a collection has a unique index on `a.b`:

```
db.collection.ensureIndex( { "a.b": 1 }, { unique: true } )
```

The unique index permits the insertion of the following document into the collection if no other document in the collection has the `a.b` value of `5`:

```
db.collection.insert( { a: [ { b: 5 }, { b: 5 } ] } )
```

Unique Index and Missing Field If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document. Because of the unique constraint, MongoDB will only permit one document that lacks the indexed field. If there is more than one document without a value for the indexed field or is missing the indexed field, the index build will fail with a duplicate key error.

You can combine the unique constraint with the *sparse index* (page 507) to filter these null values from the unique index and avoid the error.

Restrictions You may not specify a unique constraint on a *hashed index* (page 504).

See also:

Create a Unique Index (page 522)

Sparse Indexes

On this page

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is "sparse" because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

To create a `sparse` index, use the `db.collection.ensureIndex()` method with the `sparse` option set to `true`. For example, the following operation in the `mongo` shell creates a sparse index on the `xmpp_id` field of the `addresses` collection:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

Note: Do not confuse sparse indexes in MongoDB with block-level[3] indexes in other databases. Think of them as dense indexes with a specific filter.

Behavior

`sparse` Index and Incomplete Results Changed in version 2.6.

[3] http://en.wikipedia.org/wiki/Database_index#Sparse_index

If a sparse index would result in an incomplete result set for queries and sort operations, MongoDB will not use that index unless a `hint()` explicitly specifies the index.

For example, the query `{ x: { $exists: false } }` will not use a sparse index on the `x` field unless explicitly hinted. See *Sparse Index On A Collection Cannot Return Complete Results* (page 508) for an example that details the behavior.

Indexes that are `sparse` by Default *2dsphere (version 2)* (page 497), *2d* (page 498), *geoHaystack* (page 499), and *text* (page 501) indexes are always `sparse`.

`sparse` Compound Indexes Sparse *compound indexes* (page 489) that only contain ascending/descending index keys will index a document as long as the document contains at least one of the keys.

For sparse compound indexes that contain a geospatial key (i.e. *2dsphere* (page 497), *2d* (page 498), or *geoHaystack* (page 499) index keys) along with ascending/descending index key(s), only the existence of the geospatial field(s) in a document determine whether the index references the document.

For sparse compound indexes that contain *text* (page 501) index keys along with ascending/descending index keys, only the existence of the `text` index field(s) determine whether the index references a document.

`sparse` and `unique` Properties An index that is both `sparse` and *unique* (page 506) prevents collection from having documents with duplicate values for a field but allows multiple documents that omit the key.

Examples

Create a Sparse Index On A Collection Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

The collection has a sparse index on the field `score`:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true } )
```

Then, the following query on the `scores` collection uses the sparse index to return the documents that have the `score` field less than (`$lt`) `90`:

```
db.scores.find( { score: { $lt: 90 } } )
```

Because the document for the userid `"newbie"` does not contain the `score` field and thus does not meet the query criteria, the query can use the sparse index to return the results:

```
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
```

Sparse Index On A Collection Cannot Return Complete Results Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

The collection has a sparse index on the field `score`:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true } )
```

Because the document for the userid `"newbie"` does not contain the `score` field, the sparse index does not contain an entry for that document.

Consider the following query to return **all** documents in the `scores` collection, sorted by the `score` field:

```
db.scores.find().sort( { score: -1 } )
```

Even though the sort is by the indexed field, MongoDB will **not** select the sparse index to fulfill the query in order to return complete results:

```
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
```

To use the sparse index, explicitly specify the index with `hint()`:

```
db.scores.find().sort( { score: -1 } ).hint( { score: 1 } )
```

The use of the index results in the return of only those documents with the `score` field:

```
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
```

See also:

`explain()` and *Analyze Query Performance* (page 117)

Sparse Index with Unique Constraint Consider a collection `scores` that contains the following documents:

```
{ "_id" : ObjectId("523b6e32fb408eea0eec2647"), "userid" : "newbie" }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid" : "abby", "score" : 82 }
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid" : "nina", "score" : 90 }
```

You could create an index with a *unique constraint* (page 506) and sparse filter on the `score` field using the following operation:

```
db.scores.ensureIndex( { score: 1 } , { sparse: true, unique: true } )
```

This index *would permit* the insertion of documents that had unique values for the `score` field *or* did not include a `score` field. Consider the following *insert operation* (page 97):

```
db.scores.insert( { "userid": "AAAAAAA", "score": 43 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 34 } )
db.scores.insert( { "userid": "CCCCCCC" } )
db.scores.insert( { "userid": "DDDDDDD" } )
```

However, the index *would not permit* the addition of the following documents since documents already exists with `score` value of `82` and `90`:

```
db.scores.insert( { "userid": "AAAAAAA", "score": 82 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 90 } )
```

8.2.3 Index Creation

On this page

MongoDB provides several options that *only* affect the creation of the index. Specify these options in a document as the second argument to the `db.collection.ensureIndex()` method. This section describes the uses of these creation options and their behavior.

Related

Some options that you can specify to `ensureIndex()` options control the *properties of the index* (page 504), which are *not* index creation options. For example, the *unique* (page 506) option affects the behavior of the index after creation.

For a detailed description of MongoDB's index types, see *Index Types* (page 486) and *Index Properties* (page 504) for related documentation.

Background Construction

By default, creating an index blocks all other operations on a database. When building an index on a collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any operation that requires a read or write lock on all databases (e.g. **listDatabases**) will wait for the foreground index build to complete.

For potentially long running index building operations, consider the `background` operation so that the MongoDB database remains available during the index building operation. For example, to create an index in the background of the `zipcode` field of the `people` collection, issue the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

By default, `background` is `false` for building MongoDB indexes.

You can combine the background option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true, sparse: true } )
```

Behavior

As of MongoDB version 2.4, a `mongod` instance can build more than one index in the background concurrently.

Changed in version 2.4: Before 2.4, a `mongod` instance could only build one background index per database at a time.

Changed in version 2.2: Before 2.2, a single `mongod` instance could only build one index at a time.

Background indexing operations run in the background so that other database operations can run while creating the index. However, the `mongo` shell session or connection where you are creating the index *will* block until the index build is complete. To continue issuing commands to the database, open another connection or `mongo` instance.

Queries will not use partially-built indexes: the index will only be usable once the index build is complete.

Note: If MongoDB is building an index in the background, you cannot perform other administrative operations involving that collection, including running `repairDatabase`, dropping the collection (i.e. `db.collection.drop()`), and running `compact`. These operations will return an error during background index builds.

Performance

The background index operation uses an incremental approach that is slower than the normal "foreground" index builds. If the index is larger than the available RAM, then the incremental process can take *much* longer than the foreground build.

If your application includes `ensureIndex()` operations, and an index *doesn't* exist for other operational concerns, building the index can have a severe impact on the performance of the database.

To avoid performance issues, make sure that your application checks for the indexes at start up using the `getIndexes()` method or the equivalent method for your driver[4] 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

Changed in version 2.6: Secondary members can now build indexes in the background. Previously all index builds on secondaries were in the foreground.

Background index operations on a *replica set secondaries* begin after the *primary* completes building the index. If MongoDB builds an index in the background on the primary, the secondaries will then build that index in the background.

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.

The amount of time required to build the index on a secondary must be within the window of the *oplog*, so that the secondary can catch up with the primary.

Indexes on secondary members in "recovering" mode are always built in the foreground to allow them to catch up as soon as possible.

See *Build Indexes on Replica Sets* (page 524) for a complete procedure for building indexes on secondaries.

Drop Duplicates

Deprecated since version 2.6: The `dropDups` option to `ensureIndex()`, `createIndex()`, and `createIndexes` is deprecated.

MongoDB cannot create a *unique index* (page 506) on a field that has duplicate values. To force the creation of a unique index, you can specify the `dropDups` option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

Important: As in all unique indexes, if a document does not have the indexed field, MongoDB will include it in the index with a "null" value.

If subsequent fields *do not* have the indexed field, and you have set `{dropDups: true}`, MongoDB will remove these documents from the collection when creating the index. If you combine `dropDups` with the *sparse* (page 507) 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.

[4] https://api.mongodb.org/

To create a unique index that drops duplicates on the `username` field of the `accounts` collection, use a command in the following form:

```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

Warning: Specifying `{ dropDups: true }` will delete data from your database. Use with extreme caution.

By default, `dropDups` is `false`.

Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index, 1 or -1.

Example

Issue the following command to create an index on `item` and `quantity`:

```
db.products.ensureIndex( { item: 1, quantity: -1 } )
```

The resulting index is named: `item_1_quantity_-1`.

Optionally, you can specify a name for an index instead of using the default name.

Example

Issue the following command to create an index on `item` and `quantity` and specify `inventory` as the index name:

```
db.products.ensureIndex( { item: 1, quantity: -1 } , { name: "inventory" } )
```

The resulting index has the name `inventory`.

To view the name of an index, use the `getIndexes()` method.

8.2.4 Index Intersection

On this page

New in version 2.6.

MongoDB can use the intersection of multiple indexes to fulfill queries. [5] In general, each index intersection involves two indexes; however, MongoDB can employ multiple/nested index intersections to resolve a query.

To illustrate index intersection, consider a collection `orders` that has the following indexes:

```
{ qty: 1 }
{ item: 1 }
```

[5] In previous versions, MongoDB could use only a single index to fulfill most queries. The exception to this is queries with `$or` clauses, which could use a single index for each `$or` clause.

MongoDB can use the intersection of the two indexes to support the following query:

```
db.orders.find( { item: "abc123", qty: { $gt: 15 } } )
```

For query plans that use index intersection, the `explain()` returns the value `Complex Plan` in the `cursor` field.

Index Prefix Intersection

With index intersection, MongoDB can use an intersection of either the entire index or the index prefix. An index prefix is a subset of a compound index, consisting of one or more keys starting from the beginning of the index.

Consider a collection `orders` with the following indexes:

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
```

To fulfill the following query which specifies a condition on both the `qty` field and the `status` field, MongoDB can use the intersection of the two indexes:

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } )
```

Index Intersection and Compound Indexes

Index intersection does not eliminate the need for creating *compound indexes* (page 489). However, because both the list order (i.e. the order in which the keys are listed in the index) and the sort order (i.e. ascending or descending), matter in *compound indexes* (page 489), a compound index may not support a query condition that does not include the *index prefix keys* (page 490) or that specifies a different sort order.

For example, if a collection `orders` has the following compound index, with the `status` field listed before the `ord_date` field:

```
{ status: 1, ord_date: -1 }
```

The compound index can support the following queries:

```
db.orders.find( { status: { $in: ["A", "P" ] } } )
db.orders.find(
   {
     ord_date: { $gt: new Date("2014-02-01") },
     status: {$in:[ "P", "A" ] }
   }
)
```

But not the following two queries:

```
db.orders.find( { ord_date: { $gt: new Date("2014-02-01") } } )
db.orders.find( { } ).sort( { ord_date: 1 } )
```

However, if the collection has two separate indexes:

```
{ status: 1 }
{ ord_date: -1 }
```

The two indexes can, either individually or through index intersection, support all four aforementioned queries.

The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system.

See also:

compound indexes (page 489), *Create Compound Indexes to Support Several Different Queries* (page 552)

Index Intersection and Sort

Index intersection does not apply when the `sort()` operation requires an index completely separate from the query predicate.

For example, the `orders` collection has the following indexes:

```
{ qty: 1 }
{ status: 1, ord_date: -1 }
{ status: 1 }
{ ord_date: -1 }
```

MongoDB cannot use index intersection for the following query with sort:

```
db.orders.find( { qty: { $gt: 10 } } ).sort( { status: 1 } )
```

That is, MongoDB does not use the `{ qty: 1 }` index for the query, and the separate `{ status: 1 }` or the `{ status: 1, ord_date: -1 }` index for the sort.

However, MongoDB can use index intersection for the following query with sort since the index `{ status: 1, ord_date: -1 }` can fulfill part of the query predicate.

```
db.orders.find( { qty: { $gt: 10 } , status: "A" } ).sort( { ord_date: -1 } )
```

8.2.5 Multikey Index Bounds

On this page

The bounds of an index scan define the portions of an index to search during a query. When multiple predicates over an index exist, MongoDB will attempt to combine the bounds for these predicates by either *intersection* or *compounding* in order to produce a scan with smaller bounds.

Intersect Bounds for Multikey Index

Bounds intersection refers to a logical conjunction (i.e. `AND`) of multiple bounds. For instance, given two bounds `[ [ 3, Infinity ] ]` and `[ [ -Infinity, 6 ] ]`, the intersection of the bounds results in `[ [ 3, 6 ] ]`.

Given an *indexed* (page 491) array field, consider a query that specifies multiple predicates on the array and can use a *multikey index* (page 491). MongoDB can intersect *multikey index* (page 491) bounds if an `$elemMatch` joins the predicates.

For example, a collection `survey` contains documents with a field `item` and an array field `ratings`:

```
{ _id: 1, item: "ABC", ratings: [ 2, 9 ] }
{ _id: 2, item: "XYZ", ratings: [ 4, 3 ] }
```

Create a *multikey index* (page 491) on the `ratings` array:

```
db.survey.ensureIndex( { ratings: 1 } )
```

The following query uses `$elemMatch` to require that the array contains at least one *single* element that matches both conditions:

```
db.survey.find( { ratings : { $elemMatch: { $gte: 3, $lte: 6 } } } )
```

Taking the predicates separately:

- the bounds for the greater than or equal to 3 predicate (i.e. `$gte: 3`) are `[ [ 3, Infinity ] ]`;
- the bounds for the less than or equal to 6 predicate (i.e. `$lte: 6`) are `[ [ -Infinity, 6 ] ]`.

Because the query uses `$elemMatch` to join these predicates, MongoDB can intersect the bounds to:

```
ratings: [ [ 3, 6 ] ]
```

If the query does *not* join the conditions on the array field with `$elemMatch`, MongoDB cannot intersect the multikey index bounds. Consider the following query:

```
db.survey.find( { ratings : { $gte: 3, $lte: 6 } } )
```

The query searches the `ratings` array for at least one element greater than or equal to 3 and at least one element less than or equal to 6. Because a single element does not need to meet both criteria, MongoDB does *not* intersect the bounds and uses either `[ [ 3, Infinity ] ]` or `[ [ -Infinity, 6 ] ]`. MongoDB makes no guarantee as to which of these two bounds it chooses.

Compound Bounds for Multikey Index

Compounding bounds refers to using bounds for multiple keys of *compound index* (page 489). For instance, given a compound index `{ a: 1, b: 1 }` with bounds on field `a` of `[ [ 3, Infinity ] ]` and bounds on field `b` of `[ [ -Infinity, 6 ] ]`, compounding the bounds results in the use of both bounds:

```
{ a: [ [ 3, Infinity ] ], b: [ [ -Infinity, 6 ] ] }
```

If MongoDB cannot compound the two bounds, MongoDB always constrains the index scan by the bound on its leading field, in this case, `a: [ [ 3, Infinity ] ]`.

Compound Index on an Array Field

Consider a compound multikey index; i.e. a *compound index* (page 489) where one of the indexed fields is an array. For example, a collection `survey` contains documents with a field `item` and an array field `ratings`:

```
{ _id: 1, item: "ABC", ratings: [ 2, 9 ] }
{ _id: 2, item: "XYZ", ratings: [ 4, 3 ] }
```

Create a *compound index* (page 489) on the `item` field and the `ratings` field:

```
db.survey.ensureIndex( { item: 1, ratings: 1 } )
```

The following query specifies a condition on both keys of the index:

```
db.survey.find( { item: "XYZ", ratings: { $gte: 3 } } )
```

Taking the predicates separately:

- the bounds for the `item: "XYZ"` predicate are `[ [ "XYZ", "XYZ" ] ]`;
- the bounds for the `ratings: { $gte: 3 }` predicate are `[ [ 3, Infinity ] ]`.

MongoDB can compound the two bounds to use the combined bounds of:

```
{ item: [ [ "XYZ", "XYZ" ] ], ratings: [ [ 3, Infinity ] ] }
```

Compound Index on Fields from an Array of Embedded Documents

If an array contains embedded documents, to index on fields contained in the embedded documents, use the *dotted field name* (page 179) in the index specification. For instance, given the following array of embedded documents:

```
ratings: [ { score: 2, by: "mn" }, { score: 9, by: "anon" } ]
```

The dotted field name for the `score` field is `"ratings.score"`.

Compound Bounds of Non-array Field and Field from an Array Consider a collection `survey2` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [ { score: 2, by: "mn" }, { score: 9, by: "anon" } ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [ { score: 5, by: "anon" }, { score: 7, by: "wv" } ]
}
```

Create a *compound index* (page 489) on the non-array field `item` as well as two fields from an array `ratings.score` and `ratings.by`:

```
db.survey2.ensureIndex( { "item": 1, "ratings.score": 1, "ratings.by": 1 } )
```

The following query specifies a condition on all three fields:

```
db.survey2.find( { item: "XYZ",  "ratings.score": { $lte: 5 }, "ratings.by": "anon" } )
```

Taking the predicates separately:

- the bounds for the `item: "XYZ"` predicate are `[ [ "XYZ", "XYZ" ] ]`;
- the bounds for the `score: { $lte: 5 }` predicate are `[ [ -Infinity, 5 ] ]`;
- the bounds for the `by: "anon"` predicate are `[ "anon", "anon" ]`.

MongoDB can compound the bounds for the `item` key with *either* the bounds for `"ratings.score"` or the bounds for `"ratings.by"`, depending upon the query predicates and the index key values. MongoDB makes no guarantee as to which bounds it compounds with the `item` field. For instance, MongoDB will either choose to compound the `item` bounds with the `"ratings.score"` bounds:

```
{
  "item" : [ [ "XYZ", "XYZ" ] ],
  "ratings.score" : [ [ -Infinity, 5 ] ],
  "ratings.by" : [ [ MinKey, MaxKey ] ]
}
```

Or, MongoDB may choose to compound the `item` bounds with `"ratings.by"` bounds:

```
{
  "item" : [ [ "XYZ", "XYZ" ] ],
  "ratings.score" : [ [ MinKey, MaxKey ] ],
```

```
  "ratings.by" : [ [ "anon", "anon" ] ]
}
```

However, to compound the bounds for `"ratings.score"` with the bounds for `"ratings.by"`, the query must use `$elemMatch`. See *Compound Bounds of Index Fields from an Array* (page 517) for more information.

Compound Bounds of Index Fields from an Array To compound together the bounds for index keys from the same array:

- the index keys must share the same field path up to but excluding the field names, and
- the query must specify predicates on the fields using `$elemMatch` on that path.

For a field in an embedded document, the *dotted field name* (page 179), such as `"a.b.c.d"`, is the field path for `d`. To compound the bounds for index keys from the same array, the `$elemMatch` must be on the path up to *but excluding* the field name itself; i.e. `"a.b.c"`.

For instance, create a *compound index* (page 489) on the `ratings.score` and the `ratings.by` fields:

```
db.survey2.ensureIndex( { "ratings.score": 1, "ratings.by": 1 } )
```

The fields `"ratings.score"` and `"ratings.by"` share the field path `ratings`. The following query uses `$elemMatch` on the field `ratings` to require that the array contains at least one *single* element that matches both conditions:

```
db.survey2.find( { ratings: { $elemMatch: { score: { $lte: 5 }, by: "anon" } } } )
```

Taking the predicates separately:

- the bounds for the `score: { $lte: 5 }` predicate is `[ -Infinity, 5 ]`;
- the bounds for the `by: "anon"` predicate is `[ "anon", "anon" ]`.

MongoDB can compound the two bounds to use the combined bounds of:

```
{ "ratings.score" : [ [ -Infinity, 5 ] ], "ratings.by" : [ [ "anon", "anon" ] ] }
```

Query Without `$elemMatch` If the query does *not* join the conditions on the indexed array fields with `$elemMatch`, MongoDB *cannot* compound their bounds. Consider the following query:

```
db.survey2.find( { "ratings.score": { $lte: 5 }, "ratings.by": "anon" } )
```

Because a single embedded document in the array does not need to meet both criteria, MongoDB does *not* compound the bounds. When using a compound index, if MongoDB cannot constrain all the fields of the index, MongoDB always constrains the leading field of the index, in this case `"ratings.score"`:

```
{
  "ratings.score": [ [ -Infinity, 5 ] ],
  "ratings.by": [ [ MinKey, MaxKey ] ]
}
```

`$elemMatch` on Incomplete Path If the query does not specify `$elemMatch` on the path of the embedded fields, up to but excluding the field names, MongoDB **cannot** compound the bounds of index keys from the same array.

For example, a collection `survey3` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [ { score: { q1: 2, q2: 5 } }, { score: { q1: 8, q2: 4 } } ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [ { score: { q1: 7, q2: 8 } }, { score: { q1: 9, q2: 5 } } ]
}
```

Create a *compound index* (page 489) on the `ratings.score.q1` and the `ratings.score.q2` fields:

```
db.survey3.ensureIndex( { "ratings.score.q1": 1, "ratings.score.q2": 1 } )
```

The fields `"ratings.score.q1"` and `"ratings.score.q2"` share the field path `"ratings.score"` and the `$elemMatch` must be on that path.

The following query, however, uses an `$elemMatch` but not on the required path:

```
db.survey3.find( { ratings: { $elemMatch: { 'score.q1': 2, 'score.q2': 8 } } } )
```

As such, MongoDB **cannot** compound the bounds, and the `"ratings.score.q2"` field will be unconstrained during the index scan. To compound the bounds, the query must use `$elemMatch` on the path `"ratings.score"`:

```
db.survey3.find( { 'ratings.score': { $elemMatch: { 'q1': 2, 'q2': 8 } } } )
```

Compound `$elemMatch` Clauses Consider a query that contains multiple `$elemMatch` clauses on different field paths, for instance, `"a.b": { $elemMatch: ... }, "a.c": { $elemMatch: ... }`. MongoDB cannot combine the bounds of the `"a.b"` with the bounds of `"a.c"` since `"a.b"` and `"a.c"` also require `$elemMatch` on the path `a`.

For example, a collection `survey4` contains documents with a field `item` and an array field `ratings`:

```
{
  _id: 1,
  item: "ABC",
  ratings: [
    { score: { q1: 2, q2: 5 }, certainty: { q1: 2, q2: 3 } },
    { score: { q1: 8, q2: 4 }, certainty: { q1: 10, q2: 10 } }
  ]
}
{
  _id: 2,
  item: "XYZ",
  ratings: [
    { score: { q1: 7, q2: 8 }, certainty: { q1: 5, q2: 5 } },
    { score: { q1: 9, q2: 5 }, certainty: { q1: 7, q2: 7 } }
  ]
}
```

Create a *compound index* (page 489) on the `ratings.score.q1` and the `ratings.score.q2` fields:

```
db.survey4.ensureIndex( {
   "ratings.score.q1": 1,
   "ratings.score.q2": 1,
   "ratings.certainty.q1": 1,
   "ratings.certainty.q2": 1
} )
```

Consider the following query with two `$elemMatch` clauses:

```
db.survey4.find(
  {
    "ratings.score": { $elemMatch: { q1: 5, q2: 5 } },
    "ratings.certainty": { $elemMatch: { q1: 7, q2: 7 } },
  }
)
```

Taking the predicates separately:

- the bounds for the `"ratings.score"` predicate are the compound bounds:

  ```
  { "ratings.score.q1" : [ [ 5, 5 ] ], "ratings.score.q2" : [ [ 5, 5 ] ] }
  ```

- the bounds for the `"ratings.certainty"` predicate are the compound bounds:

  ```
  { "ratings.certainty.q1" : [ [ 7, 7 ] ], "ratings.certainty.q2" : [ [ 7, 7 ] ] }
  ```

However, MongoDB cannot compound the bounds for `"ratings.score"` and `"ratings.certainty"` since `$elemMatch` does not join the two. Instead, MongoDB constrains the leading field of the index `"ratings.score.q1"` which can be compounded with the bounds for `"ratings.score.q2"`:

```
{
  "ratings.score.q1" : [ [ 5, 5 ] ],
  "ratings.score.q2" : [ [ 5, 5 ] ],
  "ratings.certainty.q1" : [ [ MinKey, MaxKey ] ],
  "ratings.certainty.q2" : [ [ MinKey, MaxKey ] ]
}
```

8.3 Indexing Tutorials

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

The documents in this section outline specific tasks related to building and maintaining indexes for data in MongoDB collections and discusses strategies and practical approaches. For a conceptual overview of MongoDB indexing, see the *Index Concepts* (page 485) document.

Index Creation Tutorials **(page 519)** Create and configure different types of indexes for different purposes.

Index Management Tutorials **(page 528)** Monitor and assess index performance and rebuild indexes as needed.

Geospatial Index Tutorials **(page 533)** Create indexes that support data stored as *GeoJSON* objects and legacy coordinate pairs.

Text Search Tutorials **(page 543)** Build and configure indexes that support full-text searches.

Indexing Strategies **(page 551)** The factors that affect index performance and practical approaches to indexing in MongoDB

8.3.1 Index Creation Tutorials

Instructions for creating and configuring indexes in MongoDB and building indexes on replica sets and sharded clusters.

Create an Index **(page 520)** Build an index for any field on a collection.

Create a Compound Index **(page 521)** Build an index of multiple fields on a collection.

Create a Unique Index **(page 522)** Build an index that enforces unique values for the indexed field or fields.

Create a Sparse Index **(page 523)** Build an index that omits references to documents that do not include the indexed field. This saves space when indexing fields that are present in only some documents.

Create a Hashed Index **(page 524)** Compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

Build Indexes on Replica Sets **(page 524)** To build indexes on a replica set, you build the indexes separately on the primary and the secondaries, as described here.

Build Indexes in the Background **(page 526)** Background index construction allows read and write operations to continue while building the index, but take longer to complete and result in a larger index.

Build Old Style Indexes **(page 527)** A `{v : 0}` index is necessary if you need to roll back from MongoDB version 2.0 (or later) to MongoDB version 1.8.

Create an Index

On this page

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a *collection*. Users can create indexes for any collection on any field in a *document*. By default, MongoDB creates an index on the `_id` field of every collection.

This tutorial describes how to create an index on a single field. MongoDB also supports *compound indexes* (page 489), which are indexes on multiple fields. See *Create a Compound Index* (page 521) for instructions on building compound indexes.

Create an Index on a Single Field

To create an index, use `ensureIndex()` or a similar method from your driver[6]. The `ensureIndex()` method only creates an index if an index of the same specification does not already exist.

For example, the following operation creates an index on the `userid` field of the `records` collection:

```
db.records.ensureIndex( { userid: 1 } )
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders items in ascending order. A value of `-1` specifies an index that orders items in descending order. For additional index types, see *Index Types* (page 486).

The created index will support queries that select on the field `userid`, such as the following:

```
db.records.find( { userid: 2 } )
db.records.find( { userid: { $gt: 10 } } )
```

But the created index does not support the following query on the `profile_url` field:

```
db.records.find( { profile_url: 2 } )
```

For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

[6] https://api.mongodb.org/

Additional Considerations

Although indexes can improve query performances, indexes also present some operational considerations. See *Operational Considerations for Indexes* (page 155) for more information.

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 510). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 524) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 524).

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 521), *Indexing Tutorials* (page 519) and *Index Concepts* (page 485) for more information.

Create a Compound Index

On this page

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 489) 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 489) use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

The value of the field in the index specification describes the kind of index for that field. For example, a value of `1` specifies an index that orders items in ascending order. A value of `-1` specifies an index that orders items in descending order. For additional index types, see *Index Types* (page 486).

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 510). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 524) section for more information.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 524).

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 520), *Indexing Tutorials* (page 519) and *Index Concepts* (page 485) for more information.

Create a Unique Index

On this page

MongoDB allows you to specify a *unique constraint* (page 506) 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 choose to combine unique enforcement with *duplicate dropping* (page 511).

Unique Indexes

To create a *unique index* (page 506), 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 487) is a unique index. In some situations you may consider using the `_id` field itself for this kind of data rather than using a unique index on another field.

If a document does not have a value for a field, the index entry for that item will be `null` in any index that includes it. Thus, in many situations you will want to combine the `unique` constraint with the `sparse` option. `Sparse` indexes skip over any document that is missing the indexed field, rather than storing `null` for the index entry. 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 489), 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

Deprecated since version 2.6: The `dropDups` option to `ensureIndex()`, `createIndex()`, and `createIndexes` is deprecated.

To force the creation of a *unique index* (page 506) index on a collection with duplicate values in the field you are indexing you can use the `dropDups` option. This will force MongoDB to create a *unique* index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of `ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { unique: true, dropDups: true } )
```

See the full documentation of *duplicate dropping* (page 511) for more information.

Warning: Specifying `{ dropDups: true }` may delete data from your database. Use with extreme caution.

Refer to the `ensureIndex()` documentation for additional index creation options.

Create a Sparse Index

On this page

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 507) for more information about sparse indexes and their use.

See also:

Index Concepts (page 485) and *Indexing Tutorials* (page 519) for more information.

Prototype

To create a *sparse index* (page 507) 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 507) section for more information.

Create a Hashed Index

On this page

New in version 2.4.

Hashed indexes (page 504) compute a hash of the value of a field in a collection and index the hashed value. These indexes permit equality queries and may be suitable shard keys for some collections.

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

See

Hashed Shard Keys (page 689) for more information about hashed indexes in sharded clusters, as well as *Index Concepts* (page 485) and *Indexing Tutorials* (page 519) for more information about indexes.

Procedure

To create a *hashed index* (page 504), specify `hashed` as the value of the index key, as in the following example:

Example

Specify a hashed index on `_id`

```
db.collection.ensureIndex( { _id: "hashed" } )
```

Considerations

MongoDB supports `hashed` indexes of any single field. The hashing function collapses embedded documents and computes the hash for the entire value, but does not support multi-key (i.e. arrays) indexes.

You may not create compound indexes that have `hashed` index fields.

Build Indexes on Replica Sets

On this page

For replica sets, secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the `mongos` will send `ensureIndex()` to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.

To minimize the impact of building an index on your replica set, use the following procedure to build indexes:

See

Indexing Tutorials (page 519) and *Index Concepts* (page 485) for more information.

Considerations

- 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 597) documentation for additional information.
- 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.
- Do **not** use this procedure when building a *unique index* (page 506) with the `dropDups` option.
- Before version 2.6 *Background index creation operations* (page 510) become *foreground* indexing operations on *secondary* members of replica sets. After 2.6, background index builds replicate as background index builds on the secondaries.

Procedure

Note: If you need to build an index in a *sharded cluster*, repeat the following procedure for each replica set that provides each *shard*.

Stop One Secondary Stop the `mongod` process on one secondary. Restart the `mongod` process *without* the `--replSet` option and running on a different port. [7] This instance is now in "standalone" mode.

For example, if your `mongod` *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()` in the `mongo` shell, or comparable method in your driver. This operation will create or rebuild the index on this `mongod` instance

For example, to create an ascending index on the `username` field of the `records` collection, use the following `mongo` shell operation:

[7] By running the `mongod` on a different port, you ensure that the other members of the replica set and all clients will not contact the member while you are building the index.

```
db.records.ensureIndex( { username: 1 } )
```

See also:

Create an Index (page 520) and *Create a Compound Index* (page 521) for more information.

Restart the Program `mongod` When the index build completes, start the `mongod` 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 Changed in version 2.6: Secondary members can now *build indexes in the background* (page 526). Previously all index builds on secondaries were in the foreground.

For each secondary in the set, build an index according to the following steps:

1. *Stop One Secondary* (page 525)
2. *Build the Index* (page 525)
3. *Restart the Program mongod* (page 526)

Build the Index on the Primary To build an index on the primary you can either:

1. *Build the index in the background* (page 526) on the primary.
2. Step down the primary using the `rs.stepDown()` method in the `mongo` shell to cause the current primary to become a secondary graceful and allow the set to elect another member as primary.

   Then repeat the index building procedure, listed below, to build the index on the primary:

   (a) *Stop One Secondary* (page 525)

   (b) *Build the Index* (page 525)

   (c) *Restart the Program mongod* (page 526)

Building the index in the background takes longer than the foreground index build and results in a less compact index structure. Additionally, the background index build may impact write performance on the primary. However, building the index in the background allows the set to be continuously up for write operations while MongoDB builds the index.

Build Indexes in the Background

On this page

By default, MongoDB builds indexes in the foreground, which prevents all read and write operations to the database while the index builds. Also, no operation that requires a read or write lock on all databases (e.g. **listDatabases**) can occur during a foreground index build.

Background index construction (page 510) allows read and write operations to continue while building the index.

See also:

Index Concepts (page 485) and *Indexing Tutorials* (page 519) for more information.

Considerations

Background index builds take longer to complete and result in an index that is *initially* larger, or less compact, than an index built in the foreground. Over time, the compactness of indexes built in the background will approach foreground-built indexes.

After MongoDB finishes building the index, background-built indexes are functionally identical to any other index.

Procedure

To create an index in the background, add the `background` argument to the `ensureIndex()` operation, as in the following index:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on *background index construction* (page 510) for more information about these indexes and their implications.

Build Old Style Indexes

Important: Use this procedure *only* if you **must** have indexes that are compatible with a version of MongoDB earlier than 2.0.

MongoDB version 2.0 introduced the `{v:1}` index format. MongoDB versions 2.0 and later support both the `{v:1}` format and the earlier `{v:0}` format.

MongoDB versions prior to 2.0, however, support only the `{v:0}` format. If you need to roll back MongoDB to a version prior to 2.0, you must *drop* and *re-create* your indexes.

To build pre-2.0 indexes, use the `dropIndexes()` and `ensureIndex()` methods. You *cannot* simply reindex the collection. When you reindex on versions that only support `{v:0}` indexes, the `v` fields in the index definition still hold values of `1`, even though the indexes would now use the `{v:0}` format. If you were to upgrade again to version 2.0 or later, these indexes would not work.

Example

Suppose you rolled back from MongoDB 2.0 to MongoDB 1.8, and suppose you had the following index on the `items` collection:

```
{ "v" : 1, "key" : { "name" : 1 }, "ns" : "mydb.items", "name" : "name_1" }
```

The `v` field tells you the index is a `{v:1}` index, which is incompatible with version 1.8.

To drop the index, issue the following command:

```
db.items.dropIndex( { name : 1 } )
```

To recreate the index as a `{v:0}` index, issue the following command:

```
db.foo.ensureIndex( { name : 1 } , { v : 0 } )
```

See also:

Index Performance Enhancements (page 893).

8.3.2 Index Management Tutorials

Instructions for managing indexes and assessing index performance and use.

Remove Indexes **(page 528)** Drop an index from a collection.

Modify an Index **(page 529)** Modify an existing index.

Rebuild Indexes **(page 530)** In a single operation, drop all indexes on a collection and then rebuild them.

Manage In-Progress Index Creation **(page 531)** Check the status of indexing progress, or terminate an ongoing index build.

Return a List of All Indexes **(page 531)** Obtain a list of all indexes on a collection or of all indexes on all collections in a database.

Measure Index Use **(page 532)** Study query operations and observe index use for your database.

Remove Indexes

On this page

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 530) document.

See also:

Indexing Tutorials (page 519) and *Index Concepts* (page 485) for more information about indexes and indexing operations in MongoDB.

Remove a Specific Index

To remove an index, use the `db.collection.dropIndex()` method.

For example, the following operation removes an ascending index on the `tax-id` field in the `accounts` collection:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

The operation returns a document with the status of the operation:

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index.

For *text* (page 501) indexes, pass the index name to the `db.collection.dropIndex()` method. See *Use the Index Name to Drop a text Index* (page 547) for details.

Remove All Indexes

You can also use the `db.collection.dropIndexes()` to remove *all* indexes, except for the *_id index* (page 487) from a collection.

These shell helpers provide wrappers around the `dropIndexes` *database command*. Your `client library` may have a different or additional interface for these operations.

Modify an Index

To modify an existing index, you need to drop and recreate the index.

Step 1: Create a unique index.

Use the `ensureIndex()` method create a unique index.

```
db.orders.ensureIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 },
   { unique: true }
)
```

The method returns a document with the status of the results. The method only creates an index if the index does not already exist. See *Create an Index* (page 520) and *Index Creation Tutorials* (page 519) for more information on creating indexes.

Step 2: Attempt to modify the index.

To modify an existing index, you **cannot** just re-issue the `ensureIndex()` method with the updated specification of the index.

For example, the following operation attempts to remove the `unique` constraint from the previously created index by using the `ensureIndex()` method.

```
db.orders.ensureIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The status document returned by the operation shows an error.

Step 3: Drop the index.

To modify the index, you must drop the index first.

```
db.orders.dropIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The method returns a document with the status of the operation. Upon successful operation, the `ok` field in the returned document should specify a `1`. See *Remove Indexes* (page 528) for more information about dropping indexes.

Step 4: Recreate the index without the `unique` constraint.

Recreate the index without the `unique` constraint.

```
db.orders.ensureIndex(
   { "cust_id" : 1, "ord_date" : -1, "items" : 1 }
)
```

The method returns a document with the status of the results. Upon successful operation, the returned document should show the `numIndexesAfter` to be greater than `numIndexesBefore` by one.

See also:

Index Introduction (page 481), *Index Concepts* (page 485).

Rebuild Indexes

On this page

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the *_id index* (page 487), and then rebuilds all indexes.

See also:

Index Concepts (page 485) and *Indexing Tutorials* (page 519).

Process

The operation takes the following form:

```
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        "indexes" : [
                {
                        "key" : {
                                "_id" : 1,
                                "tax-id" : 1
                        },
                        "ns" : "records.accounts",
                        "name" : "_id_"
                }
        ],
        "ok" : 1
}
```

This shell helper provides a wrapper around the `reIndex` *database command*. Your `client library` 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 524).

Manage In-Progress Index Creation

On this page

View Index Creation Operations

To see the status of an indexing process, you can use the `db.currentOp()` method in the `mongo` shell. To filter the current operations for index creation operations, see *currentOp-index-creation* for an example.

The `msg` field will include the percent of the build that is complete.

Terminate Index Creation

To terminate an ongoing index build, use the `db.killOp()` method in the `mongo` shell. For index builds, the effects of `db.killOp()` may not be immediate and may occur well after much of the index build operation has completed.

You cannot terminate a *replicated* index build on secondary members of a replica set. To minimize the impact of building an index on replica sets, see *Build Indexes on Replica Sets* (page 524).

Changed in version 2.4: Before MongoDB 2.4, you could *only* terminate *background* index builds. After 2.4, you can terminate both *background* index builds and foreground index builds.

See also:

`db.currentOp()`, `db.killOp()`

Return a List of All Indexes

On this page

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 304) collection, and you can use standard queries (i.e. `find()`) to list the indexes, or in the `mongo` shell, the `getIndexes()` method to return a list of the indexes on a collection, as in the following examples.

See also:

Index Concepts (page 485) and *Indexing Tutorials* (page 519) for more information about indexes in MongoDB and common index management operations.

List all Indexes on a Collection

To return a list of all indexes on a collection, use the `db.collection.getIndexes()` method or a similar method for your driver[8].

For example, to view all indexes on the `people` collection:

```
db.people.getIndexes()
```

List all Indexes for a Database

To return a list of all indexes on all collections in a database, use the following operation in the `mongo` shell:

```
db.system.indexes.find()
```

See `system.indexes` (page 304) for more information about these documents.

Measure Index Use

On this page

Synopsis

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides a number of tools that allow you to study query operations and observe index use for your database.

See also:

Index Concepts (page 485) and *Indexing Tutorials* (page 519) for more information.

Operations

Return Query Plan with `explain()` Append the `explain()` method to any cursor (e.g. query) to return a document with statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Control Index Use with `hint()` Append the `hint()` to any cursor (e.g. query) with the index as the argument to *force* MongoDB to use a specific index to fulfill the query. Consider the following example:

```
db.people.find( { name: "John Doe", zipcode: { $gt: "63000" } } ).hint( { zipcode: 1 } )
```

You can use `hint()` and `explain()` in conjunction with each other to compare the effectiveness of a specific index. Specify the `$natural` operator to the `hint()` method to prevent MongoDB from using *any* index:

```
db.people.find( { name: "John Doe", zipcode: { $gt: "63000" } } ).hint( { $natural: 1 } )
```

[8] https://api.mongodb.org/

Instance Index Use Reporting MongoDB provides a number of metrics of index use and operation that you may want to consider when analyzing index use for your database:

- In the output of `serverStatus`:
  - `indexCounters`
  - `scanned`
  - `scanAndOrder`
- In the output of `collStats`:
  - `totalIndexSize`
  - `indexSizes`
- In the output of `dbStats`:
  - `dbStats.indexes`
  - `dbStats.indexSize`

8.3.3 Geospatial Index Tutorials

Instructions for creating and querying `2d`, `2dsphere`, and haystack indexes.

Create a 2dsphere Index **(page 533)** A `2dsphere` index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

Query a 2dsphere Index **(page 535)** Search for locations within, near, or intersected by a GeoJSON shape, or within a circle as defined by coordinate points on a sphere.

Create a 2d Index **(page 537)** Create a `2d` index to support queries on data stored as legacy coordinate pairs.

Query a 2d Index **(page 538)** Search for locations using legacy coordinate pairs.

Create a Haystack Index **(page 540)** A haystack index is optimized to return results over small areas. For queries that use spherical geometry, a `2dsphere` index is a better option.

Query a Haystack Index **(page 540)** Search based on location and non-location data within a small area.

Calculate Distance Using Spherical Geometry **(page 541)** Convert distances to radians and back again.

Create a `2dsphere` Index

On this page

To create a geospatial index for GeoJSON-formatted data, use the `db.collection.ensureIndex()` method to create a *2dsphere index* (page 497). In the index specification document for the `db.collection.ensureIndex()` method, specify the location field as the index key and specify the string literal `"2dsphere"` as the value:

```
db.collection.ensureIndex( { <location field> : "2dsphere" } )
```

The following procedure presents steps to populate a collection with documents that contain a GeoJSON data field and create *2dsphere indexes* (page 497). Although the procedure populates the collection first, you can also create the indexes before populating the collection.

Procedure

First, populate a collection `places` with documents that store location data as *GeoJSON Point* (page 558) in a field named `loc`. The coordinate order is longitude, then latitude.

```
db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
      name: "Central Park",
      category : "Parks"
   }
)

db.places.insert(
   {
      loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },
      name: "La Guardia Airport",
      category : "Airport"
   }
)
```

Then, create the *2dsphere* (page 497) index.

Create a `2dsphere` Index For example, the following creates a *2dsphere* (page 497) index on the location field `loc`:

```
db.places.ensureIndex( { loc : "2dsphere" } )
```

Create a Compound Index with `2dsphere` Index Key A *compound index* (page 489) can include a `2dsphere` index key in combination with non-geospatial index keys. For example, the following operation creates a compound index where the first key `loc` is a `2dsphere` index key, and the remaining keys `category` and `names` are non-geospatial index keys, specifically descending (`-1`) and ascending (`1`) keys respectively.

```
db.places.ensureIndex( { loc : "2dsphere" , category : -1, name: 1 } )
```

Unlike the *2d* (page 498) index, a compound `2dsphere` index does not require the location field to be the first field indexed. For example:

```
db.places.ensureIndex( { category : 1 , loc : "2dsphere" } )
```

Considerations

Fields with *2dsphere* (page 497) indexes must hold geometry data in the form of *coordinate pairs* or *GeoJSON* data. If you attempt to insert a document with non-geometry data in a `2dsphere` indexed field, or build a `2dsphere` index on a collection where the indexed field has non-geometry data, the operation will fail.

The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2dsphere` index and/or only one *2d* (page 498) index whereas *geospatial query operators* (e.g. `$near` and `$geoWithin`) permit collections to have multiple geospatial indexes.

The geospatial index restriction for the `geoNear` command and the `$geoNear` pipeline stage exists because neither the `geoNear` command nor the `$geoNear` pipeline stage syntax includes the location field. As such, index selection among multiple `2d` indexes or `2dsphere` indexes is ambiguous.

No such restriction applies for *geospatial query operators* since these operators take a location field, eliminating the ambiguity.

As such, although this tutorial creates multiple `2dsphere` indexes, to use the `geoNear` command or the `$geoNear` pipeline stage against the example collection, you will need to `drop` all but one of the `2dsphere` indexes.

To query using the `2dsphere` index, see *Query a 2dsphere Index* (page 535).

Query a `2dsphere` Index

On this page

The following sections describe queries supported by the `2dsphere` index.

GeoJSON Objects Bounded by a Polygon

The `$geoWithin` operator queries for location data found within a GeoJSON polygon. Your location data must be stored in GeoJSON format. Use the following syntax:

```
db.<collection>.find( { <location field> :
                          { $geoWithin :
                            { $geometry :
                              { type : "Polygon" ,
                                coordinates : [ <coordinates> ]
                        } } } } )
```

The following example selects all points and shapes that exist entirely within a GeoJSON polygon:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates : [ [
                                          [ 0 , 0 ] ,
                                          [ 3 , 6 ] ,
                                          [ 6 , 1 ] ,
                                          [ 0 , 0 ]
                                        ] ]
                } } } } )
```

Intersections of GeoJSON Objects

New in version 2.4.

The `$geoIntersects` operator queries for locations that intersect a specified GeoJSON object. A location intersects the object if the intersection is non-empty. This includes documents that have a shared edge.

The `$geoIntersects` operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                          { $geoIntersects :
                            { $geometry :
```

```
                                { type : "<GeoJSON object type>" ,
                                  coordinates : [ <coordinates> ]
                        } } } } )
```

The following example uses `$geoIntersects` to select all indexed points and shapes that intersect with the polygon defined by the `coordinates` array.

```
db.places.find( { loc :
                  { $geoIntersects :
                    { $geometry :
                      { type : "Polygon" ,
                        coordinates: [ [
                                         [ 0 , 0 ] ,
                                         [ 3 , 6 ] ,
                                         [ 6 , 1 ] ,
                                         [ 0 , 0 ]
                                       ] ]
                  } } } } )
```

Proximity to a GeoJSON Point

Proximity queries return the points closest to the defined point and sorts the results by distance. A proximity query on GeoJSON data requires a `2dsphere` index.

To query for proximity to a GeoJSON point, use either the `$near` operator or `geoNear` command. Distance is in meters.

The `$near` uses the following syntax:

```
db.<collection>.find( { <location field> :
                         { $near :
                           { $geometry :
                              { type : "Point" ,
                                coordinates : [ <longitude> , <latitude> ] } ,
                             $maxDistance : <distance in meters>
                        } } } )
```

For examples, see `$near`.

The `geoNear` command uses the following syntax:

```
db.runCommand( { geoNear : <collection> ,
                 near : { type : "Point" ,
                          coordinates: [ <longitude>, <latitude> ] } ,
                 spherical : true } )
```

The `geoNear` command offers more options and returns more information than does the `$near` operator. To run the command, see `geoNear`.

Points within a Circle Defined on a Sphere

To select all grid coordinates in a "spherical cap" on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distance Using Spherical Geometry* (page 541).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                          { $centerSphere :
                             [ [ <x>, <y> ] , <radius> ] }
                      } } )
```

The following example queries grid coordinates and returns all documents within a 10 mile radius of longitude `88 W` and latitude `30 N`. The example converts the distance, 10 miles, to radians by dividing by the approximate radius of the earth, 3959 miles:

```
db.places.find( { loc :
                  { $geoWithin :
                    { $centerSphere :
                       [ [ -88 , 30 ] , 10 / 3959 ]
                } } } )
```

Create a `2d` Index

On this page

To build a geospatial `2d` index, use the `ensureIndex()` method and specify `2d`. Use the following syntax:

```
db.<collection>.ensureIndex( { <location field> : "2d" ,
                               <additional field> : <value> } ,
                             { <index-specification options> } )
```

The `2d` index uses the following optional index-specification options:

```
{ min : <lower bound> , max : <upper bound> ,
  bits : <bit precision> }
```

Define Location Range for a `2d` Index

By default, a `2d` index assumes longitude and latitude and has boundaries of -180 **inclusive** and 180 **non-inclusive**. 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 500).

Query a 2d Index

On this page

The following sections describe queries supported by the 2d index.

Points within a Shape Defined on a Flat Surface

To select all legacy coordinate pairs found within a given shape on a flat surface, use the $geoWithin operator along with a shape operator. Use the following syntax:

```
db.<collection>.find( { <location field> :
                        { $geoWithin :
                           { $box|$polygon|$center : <coordinates>
                     } } } )
```

The following queries for documents within a rectangle defined by [0 , 0] at the bottom left corner and by [100 , 100] at the top right corner.

```
db.places.find( { loc :
                  { $geoWithin :
                     { $box : [ [ 0 , 0 ] ,
                                [ 100 , 100 ] ]
                  } } } )
```

The following queries for documents that are within the circle centered on [-74 , 40.74] and with a radius of 10:

```
db.places.find( { loc: { $geoWithin :
                         { $center : [ [-74, 40.74 ] , 10 ]
                } } } )
```

For syntax and examples for each shape, see the following:

- `$box`
- `$polygon`
- `$center` (defines a circle)

Points within a Circle Defined on a Sphere

MongoDB supports rudimentary spherical queries on flat `2d` indexes for legacy reasons. In general, spherical calculations should use a `2dsphere` index, as described in *2dsphere Indexes* (page 497).

To query for legacy coordinate pairs in a "spherical cap" on a sphere, use `$geoWithin` with the `$centerSphere` operator. Specify an array that contains:

- The grid coordinates of the circle's center point
- The circle's radius measured in radians. To calculate radians, see *Calculate Distance Using Spherical Geometry* (page 541).

Use the following syntax:

```
db.<collection>.find( { <location field> :
                          { $geoWithin :
                             { $centerSphere : [ [ <x>, <y> ] , <radius> ] }
                       } } )
```

The following example query returns all documents within a 10-mile radius of longitude `88 W` and latitude `30 N`. The example converts distance to radians by dividing distance by the approximate radius of the earth, 3959 miles:

```
db.<collection>.find( { loc : { $geoWithin :
                                  { $centerSphere :
                                     [ [ 88 , 30 ] , 10 / 3959 ]
                       } } } )
```

Proximity to a Point on a Flat Surface

Proximity queries return the 100 legacy coordinate pairs closest to the defined point and sort the results by distance. Use either the `$near` operator or `geoNear` command. Both require a `2d` index.

The `$near` operator uses the following syntax:

```
db.<collection>.find( { <location field> :
                          { $near : [ <x> , <y> ]
                       } } )
```

For examples, see `$near`.

The `geoNear` command uses the following syntax:

```
db.runCommand( { geoNear: <collection>, near: [ <x> , <y> ] } )
```

The `geoNear` command offers more options and returns more information than does the `$near` operator. To run the command, see `geoNear`.

Exact Matches on a Flat Surface

Changed in version 2.6: Previously, `2d` indexes would support exact-match queries for coordinate pairs.

You cannot use a `2d` index to return an exact match for a coordinate pair. Use a scalar, ascending or descending, index on a field that stores coordinates to return exact matches.

In the following example, the `find()` operation will return an exact match on a location if you have a `{ 'loc': 1}` index:

```
db.<collection>.find( { loc: [ <x> , <y> ] } )
```

This query will return any documents with the value of `[ <x> , <y> ]`.

Create a Haystack Index

A haystack index must 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> } )
```

To build a haystack index, you must specify 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.

Example

If you have a collection with documents that contain fields similar to the following:

```
{ _id : 100, pos: { lng : 126.9, lat : 35.2 } , type : "restaurant"}
{ _id : 200, pos: { lng : 127.5, lat : 36.1 } , type : "restaurant"}
{ _id : 300, pos: { lng : 128.0, lat : 36.7 } , type : "national park"}
```

The following operations create a haystack index with buckets that store keys within 1 unit of longitude or latitude.

```
db.places.ensureIndex( { pos : "geoHaystack", type : 1 } ,
                       { bucketSize : 1 } )
```

This index stores the document with an `_id` field that has the value `200` in two different buckets:

- In a bucket that includes the document where the `_id` field has a value of `100`
- In a bucket that includes the document where the `_id` field has a value of `300`

To query using a haystack index you use the `geoSearch` command. See *Query a Haystack Index* (page 540).

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 540).

To query a haystack index, use the `geoSearch` command. You must specify both the coordinates and the additional field to `geoSearch`. For example, to return all documents with the value `restaurant` in the `type` field near the example point, the command would resemble:

```
db.runCommand( { geoSearch : "places" ,
                 search : { type: "restaurant" } ,
                 near : [-74, 40.74] ,
                 maxDistance : 10 } )
```

Note: Haystack indexes are not suited to queries for the complete list of documents closest to a particular location. The closest documents could be more distant compared to the bucket size.

Note: *Spherical query operations* (page 541) are not currently supported by haystack indexes.

The `find()` method and `geoNear` command cannot access the haystack index.

Calculate Distance Using Spherical Geometry

On this page

Note: While basic queries using spherical distance are supported by the `2d` index, consider moving to a `2dsphere` index if your data is primarily longitude and latitude.

The `2d` index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

- `$nearSphere`
- `$centerSphere`
- `$near`
- `geoNear` command with the `{ spherical: true }` option.

Important: These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- *distance to radians*: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- *radians to distance*: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The radius of the Earth is approximately `3,959` miles or `6,371` kilometers.

The following query would return documents from the `places` collection within the circle described by the center `[ -74, 40.74 ]` with a radius of `100` miles:

```
db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ] ,
                                                         100 / 3959 ] } } } )
```

You may also use the `distanceMultiplier` option to the `geoNear` to convert radians in the `mongod` process, rather than in your application code. See *distance multiplier* (page 542).

The following spherical query, returns all documents in the collection `places` within `100` miles from the point `[ -74, 40.74 ]`.

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true
               } )
```

The output of the above command would be:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 0.01853688938212826,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

Warning: Spherical queries that wrap around the poles or at the transition from `-180` to `180` longitude raise an error.

Note: While the default Earth-like bounds for geospatial indexes are between `-180` inclusive, and `180`, valid values for latitude are between `-90` and `90`.

Distance Multiplier

The `distanceMultiplier` option of the `geoNear` command returns distances only after multiplying the results by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Using `distanceMultiplier` in spherical queries provides results from the `geoNear` command that do not need radian-to-distance conversion. The following example uses `distanceMultiplier` in the `geoNear` command with a *spherical* (page 541) 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
}
```

8.3.4 Text Search Tutorials

Instructions for enabling MongoDB's text search feature, and for building and configuring text indexes.

Create a text Index **(page 543)** A `text` index allows searches on text strings in the index's specified fields.

Specify a Language for Text Index **(page 544)** The specified language determines the list of stop words and the rules for Text Search's stemmer and tokenizer.

Specify Name for text Index **(page 546)** Override the `text` index name limit for long index names.

Control Search Results with Weights **(page 547)** Give priority to certain search values by denoting the significance of an indexed field relative to other indexed fields

Limit the Number of Entries Scanned **(page 548)** Create an index to support queries that includes `$text` expressions and equality conditions.

Text Search in the Aggregation Pipeline **(page 549)** Perform various text search in the aggregation pipeline.

Create a `text` Index

On this page

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 use 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" }
                          )
```

Note: In order to drop a `text` index, use the index name. See *Use the Index Name to Drop a text Index* (page 547) for more information.

Specify a Language for Text Index

On this page

This tutorial describes how to *specify the default language associated with the text index* (page 544) and also how to *create text indexes for collections that contain documents in different languages* (page 545).

Specify the Default Language for a `text` Index

The default language associated with the indexed data determines the rules to parse word roots (i.e. stemming) and ignore stop words. 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 561) for the languages available for `default_language`.

The following example creates for the `quotes` collection a `text` index on the `content` field and sets the `default_language` to `spanish`:

```
db.quotes.ensureIndex(
   { content : "text" },
   { default_language: "spanish" }
)
```

Create a `text` Index for a Collection in Multiple Languages

Changed in version 2.6: Added support for language overrides within embedded documents.

Specify the Index Language within the Document If a collection contains documents or embedded documents that are in different languages, include a field named `language` in the documents or embedded documents and specify as its value the language for that document or embedded document.

MongoDB will use the specified language for that document or embedded document when building the `text` index:

- The specified language in the document overrides the default language for the `text` index.
- The specified language in an embedded document override the language specified in an enclosing document or the default language for the index.

See *Text Search Languages* (page 561) for a list of supported languages.

For example, a collection `quotes` contains multi-language documents that include the `language` field in the document and/or the embedded document as needed:

```
{
   _id: 1,
   language: "portuguese",
   original: "A sorte protege os audazes.",
   translation:
     [
        {
           language: "english",
           quote: "Fortune favors the bold."
        },
        {
           language: "spanish",
           quote: "La suerte protege a los audaces."
        }
    ]
}
{
   _id: 2,
   language: "spanish",
   original: "Nada hay más surrealista que la realidad.",
   translation:
      [
        {
          language: "english",
          quote: "There is nothing more surreal than reality."
        },
        {
          language: "french",
          quote: "Il n'y a rien de plus surréaliste que la réalité."
        }
      ]
}
```

```
{
   _id: 3,
   original: "is this a dagger which I see before me.",
   translation:
   {
      language: "spanish",
      quote: "Es este un puñal que veo delante de mí."
   }
}
```

If you create a `text` index on the `quote` field with the default language of English.

```
db.quotes.ensureIndex( { original: "text", "translation.quote": "text" } )
```

Then, for the documents and embedded documents that contain the `language` field, the `text` index uses that language to parse word stems and other linguistic characteristics.

For embedded documents that do not contain the `language` field,

- If the enclosing document contains the `language` field, then the index uses the document's language for the embedded document.
- Otherwise, the index uses the default language for the embedded documents.

For documents that do not contain the `language` field, the index uses the default language, which is English.

Use any Field to Specify the Language for a Document To use a field with a name other than `language`, include the `language_override` option when creating the index.

For example, give the following command to use `idioma` as the field name instead of `language`:

```
db.quotes.ensureIndex( { quote : "text" },
                       { language_override: "idioma" } )
```

The documents of the `quotes` collection may specify a language with the `idioma` field:

```
{ _id: 1, idioma: "portuguese", quote: "A sorte protege os audazes" }
{ _id: 2, idioma: "spanish", quote: "Nada hay más surrealista que la realidad." }
{ _id: 3, idioma: "english", quote: "is this a dagger which I see before me" }
```

Specify Name for `text` Index

On this page

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"
```

The `text` index, like other indexes, must fall within the `index name length limit`.

Specify a Name for `text` Index

To avoid creating an index with a name that exceeds the `index name length limit`, you can pass the `name` option to the `db.collection.ensureIndex()` method:

```
db.collection.ensureIndex(
   {
     content: "text",
     "users.comments": "text",
     "users.profiles": "text"
   },
   {
     name: "MyTextIndex"
   }
)
```

Use the Index Name to Drop a `text` Index

Whether the *text* (page 501) index has the default name or you specified a name for the *text* (page 501) index, to drop the *text* (page 501) index, pass the index name to the `db.collection.dropIndex()` method.

For example, consider the index created by the following operation:

```
db.collection.ensureIndex(
   {
     content: "text",
     "users.comments": "text",
     "users.profiles": "text"
   },
   {
     name: "MyTextIndex"
   }
)
```

Then, to remove this text index, pass the name `"MyTextIndex"` to the `db.collection.dropIndex()` method, as in the following:

```
db.collection.dropIndex("MyTextIndex")
```

To get the names of the indexes, use the `db.collection.getIndexes()` method.

Control Search Results with Weights

This document describes how to create a `text` index with specified weights for results fields.

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. See `$meta` operator for details on returning and sorting by text scores.

The default weight is 1 for the indexed fields. To adjust the weights for the indexed fields, include the `weights` option in the `db.collection.ensureIndex()` method.

Warning: Choose the weights carefully in order to prevent the need to reindex.

A collection `blog` has the following documents:

```
{ _id: 1,
  content: "This morning I had a cup of coffee.",
  about: "beverage",
  keywords: [ "coffee" ]
}

{ _id: 2,
  content: "Who doesn't like cake?",
  about: "food",
  keywords: [ "cake", "food", "dessert" ]
}
```

To create a `text` index with different field weights for the `content` field and the `keywords` field, include the `weights` option to the `ensureIndex()` method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```
db.blog.ensureIndex(
                      {
                        content: "text",
                        keywords: "text",
                        about: "text"
                      },
                      {
                        weights: {
                                    content: 10,
                                    keywords: 5
                                  },
                        name: "TextIndex"
                      }
                    )
```

The `text` index has the following fields and weights:

- `content` has a weight of 10,
- `keywords` has a weight of 5, and
- `about` has the default weight of 1.

These weights denote the relative significance of the indexed fields to each other. For instance, a term match in the `content` field has:

- `2` times (i.e. `10:5`) the impact as a term match in the `keywords` field and
- `10` times (i.e. `10:1`) the impact as a term match in the `about` field.

Limit the Number of Entries Scanned

This tutorial describes how to create indexes to limit the number of index entries scanned for queries that includes a `$text` expression and equality conditions.

A collection `inventory` contains the following documents:

```
{ _id: 1, dept: "tech", description: "lime green computer" }
{ _id: 2, dept: "tech", description: "wireless red mouse" }
```

```
{ _id: 3, dept: "kitchen", description: "green placemat" }
{ _id: 4, dept: "kitchen", description: "red peeler" }
{ _id: 5, dept: "food", description: "green apple" }
{ _id: 6, dept: "food", description: "red potato" }
```

Consider the common use case that performs text searches by *individual* departments, such as:

```
db.inventory.find( { dept: "kitchen", $text: { $search: "green" } } )
```

To limit the text search to scan only those documents within a specific `dept`, create a compound index that *first* specifies an ascending/descending index key on the field `dept` and then a `text` index key on the field `description`:

```
db.inventory.ensureIndex(
   {
     dept: 1,
     description: "text"
   }
)
```

Then, the text search [9] within a particular department will limit the scan of indexed documents. For example, the following query scans only those documents with `dept` equal to `kitchen`:

```
db.inventory.find( { dept: "kitchen", $text: { $search: "green" } } )
```

Note:

- A compound `text` index cannot include any other special index types, such as *multi-key* (page 491) or *geospatial* (page 495) index fields.
- If the compound `text` index includes keys **preceding** the `text` index key, to perform a `$text` search, the query predicate must include **equality match conditions** on the preceding keys.

See also:

Text Indexes (page 501)

Text Search in the Aggregation Pipeline

On this page

New in version 2.6. In the aggregation pipeline, text search is available via the use of the `$text` query operator in the `$match` stage.

Restrictions

Text search in the aggregation pipeline has the following restrictions:

[9] If using the deprecated `text` command, the `text` command **must** include the `filter` option that specifies an **equality** condition for the prefix fields.

- The $match stage that includes a $text must be the **first** stage in the pipeline.
- A text operator can only occur once in the stage.
- The text operator expression cannot appear in $or or $not expressions.
- The text search, by default, does not return the matching documents in order of matching scores. Use the $meta aggregation expression in the $sort stage.

Text Score

The $text operator assigns a score to each document that contains the search term in the indexed fields. The score represents the relevance of a document to a given text search query. The score can be part of a $sort pipeline specification as well as part of the projection expression. The { $meta: "textScore" } expression provides information on the processing of the $text operation. See $meta aggregation for details on accessing the score for projection or sort.

The metadata is only available after the $match stage that includes the $text operation.

Examples The following examples assume a collection articles that has a text index on the field subject:

```
db.articles.ensureIndex( { subject: "text" } )
```

Calculate the Total Views for Articles that Contains a Word

The following aggregation searches for the term cake in the $match stage and calculates the total views for the matching documents in the $group stage.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake" } } },
     { $group: { _id: null, views: { $sum: "$views" } } }
   ]
)
```

Return Results Sorted by Text Search Score

To sort by the text search score, include a $meta expression in the $sort stage. The following example matches on *either* the term cake or tea, sorts by the textScore in descending order, and returns only the title field in the results set.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake tea" } } },
     { $sort: { score: { $meta: "textScore" } } },
     { $project: { title: 1, _id: 0 } }
   ]
)
```

The specified metadata determines the sort order. For example, the "textScore" metadata sorts in descending order. See $meta for more information on metadata as well as an example of overriding the default sort order of the metadata.

Match on Text Score

The `"textScore"` metadata is available for projections, sorts, and conditions subsequent the `$match` stage that includes the `$text` operation.

The following example matches on *either* the term `cake` or `tea`, projects the `title` and the `score` fields, and then returns only those documents with a `score` greater than `1.0`.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "cake tea" } } },
     { $project: { title: 1, _id: 0, score: { $meta: "textScore" } } },
     { $match: { score: { $gt: 1.0 } } }
   ]
)
```

Specify a Language for Text Search

The following aggregation searches in spanish for documents that contain the term `saber` but not the term `claro` in the `$match` stage and calculates the total `views` for the matching documents in the `$group` stage.

```
db.articles.aggregate(
   [
     { $match: { $text: { $search: "saber -claro", $language: "es" } } },
     { $group: { _id: null, views: { $sum: "$views" } } }
   ]
)
```

8.3.5 Indexing Strategies

The best indexes for your application must take a number of factors into account, including the kinds of queries you expect, the ratio of reads to writes, and the amount of free memory on your system.

When developing your indexing strategy you should have a deep understanding of your application's queries. Before you build indexes, map out the types of queries you will run so that you can build indexes that reference those fields. Indexes come with a performance cost, but are more than worth the cost for frequent queries on large data set. Consider the relative frequency of each query in the application and whether the query justifies an index.

The best overall strategy for designing indexes is to profile a variety of index configurations with data sets similar to the ones you'll be running in production to see which configurations perform best.Inspect the current indexes created for your collections to ensure they are supporting your current and planned queries. If an index is no longer used, drop the index.

Generally, MongoDB only uses *one* index to fulfill most queries. However, each clause of an `$or` query may use a different index, and starting in 2.6, MongoDB can use an *intersection* (page 512) of multiple indexes.

The following documents introduce indexing strategies:

***Create Indexes to Support Your Queries* (page 552)** An index supports a query when the index contains all the fields scanned by the query. Creating indexes that supports queries results in greatly increased query performance.

***Use Indexes to Sort Query Results* (page 553)** To support efficient queries, use the strategies here when you specify the sequential order and sort order of index fields.

***Ensure Indexes Fit in RAM* (page 555)** When your index fits in RAM, the system can avoid reading the index from disk and you get the fastest processing.

Create Queries that Ensure Selectivity **(page 555)** Selectivity is the ability of a query to narrow results using the index. Selectivity allows MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Create Indexes to Support Your Queries

On this page

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 support 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 489) on multiple fields can support all the queries that search a "prefix" subset of those fields.

Example

The following index on a collection:

```
{ x: 1, y: 1, z: 1 }
```

Can support queries that the following indexes support:

```
{ x: 1 }
{ x: 1, y: 1 }
```

There are some situations where the prefix indexes may offer better query performance: for example if `z` is a large array.

The `{ x: 1, y: 1, z: 1 }` index can also support many of the same queries as the following index:

```
{ x: 1, z: 1 }
```

Also, `{ x: 1, z: 1 }` has an additional use. Given the following query:

```
db.collection.find( { x: 5 } ).sort( { z: 1} )
```

The `{ x: 1, z: 1 }` index supports both the query and the sort operation, while the `{ x: 1, y: 1, z: 1 }` index only supports the query. For more information on sorting, see *Use Indexes to Sort Query Results* (page 553).

Starting in version 2.6, MongoDB can use *index intersection* (page 512) to fulfill queries. The choice between creating compound indexes that support your queries or relying on index intersection depends on the specifics of your system. See *Index Intersection and Compound Indexes* (page 513) for more details.

Use Indexes to Sort Query Results

On this page

In MongoDB, sort operations can obtain the sort order by retrieving documents based on the ordering in an index. If the query planner cannot obtain the sort order from an index, it will sort the results in memory. Sort operations that use an index often have better performance than those that do not use an index. In addition, sort operations that do *not* use an index will abort when they use 32 megabytes of memory.

Sort with a Single Field Index

If an ascending or a descending index is on a single field, the sort operation on the field can be in either direction.

For example, create an ascending index on the field `a` for a collection `records`:

```
db.records.ensureIndex( { a: 1 } )
```

This index can support an ascending sort on `a`:

```
db.records.find().sort( { a: 1 } )
```

The index can also support the following descending sort on `a` by traversing the index in reverse order:

```
db.records.find().sort( { a: -1 } )
```

Sort on Multiple Fields

Create a *compound index* (page 489) to support sorting on multiple fields.

You can specify a sort on all the keys of the index or on a subset; however, the sort keys must be listed in the *same order* as they appear in the index. For example, an index key pattern `{ a: 1, b: 1 }` can support a sort on `{ a: 1, b: 1 }` but *not* on `{ b: 1, a: 1 }`.

The sort must specify the *same sort direction* (i.e.ascending/descending) for all its keys as the index key pattern or specify the *reverse sort direction* for all its keys as the index key pattern. For example, an index key pattern `{ a: 1, b: 1 }` can support a sort on `{ a: 1, b: 1 }` and `{ a: -1, b: -1 }` but *not* on `{ a: -1, b: 1 }`.

Sort and Index Prefix If the sort keys correspond to the index keys or an index *prefix*, MongoDB can use the index to sort the query results. A *prefix* of a compound index is a subset that consists of one or more keys at the start of the index key pattern.

For example, create a compound index on the `data` collection:

```
db.data.ensureIndex( { a:1, b: 1, c: 1, d: 1 } )
```

Then, the following are prefixes for that index:

```
{ a: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 1, c: 1 }
```

The following query and sort operations use the index prefixes to sort the results. These operations do not need to sort the result set in memory.

Example	Index Prefix
`db.data.find().sort( { a: 1 } )`	`{ a: 1 }`
`db.data.find().sort( { a: -1 } )`	`{ a: 1 }`
`db.data.find().sort( { a: 1, b: 1 } )`	`{ a: 1, b: 1 }`
`db.data.find().sort( { a: -1, b: -1 } )`	`{ a: 1, b: 1 }`
`db.data.find().sort( { a: 1, b: 1, c: 1 } )`	`{ a: 1, b: 1, c: 1 }`
`db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )`	`{ a: 1, b: 1 }`

Consider the following example in which the prefix keys of the index appear in both the query predicate and the sort:

```
db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )
```

In such cases, MongoDB can use the index to retrieve the documents in order specified by the sort. As the example shows, the index prefix in the query predicate can be different from the prefix in the sort.

Sort and Non-prefix Subset of an Index An index can support sort operations on a non-prefix subset of the index key pattern. To do so, the query must include **equality** conditions on all the prefix keys that precede the sort keys.

For example, the collection `data` has the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following operations can use the index to get the sort order:

Example	Index Prefix
`db.data.find( { a: 5 } ).sort( { b: 1, c: 1 } )`	`{ a: 1 , b: 1, c: 1 }`
`db.data.find( { b: 3, a: 4 } ).sort( { c: 1 } )`	`{ a: 1, b: 1, c: 1 }`
`db.data.find( { a: 5, b: { $lt: 3} } ).sort( { b: 1 } )`	`{ a: 1, b: 1 }`

As the last operation shows, only the index fields *preceding* the sort subset must have the equality conditions in the query document; the other index fields may specify other conditions.

If the query does **not** specify an equality condition on an index prefix that precedes or overlaps with the sort specification, the operation will **not** efficiently use the index. For example, 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.data.find( { a: { $gt: 2 } } ).sort( { c: 1 } )
db.data.find( { c: 5 } ).sort( { c: 1 } )
```

These operations **will not** efficiently use the index `{ a: 1, b: 1, c: 1, d: 1 }` and may not even use the index to retrieve the documents.

Ensure Indexes Fit in RAM

On this page

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` helper, which returns data in bytes:

```
> db.collection.totalIndexSize()
4294976499
```

The above example shows an index size of almost 4.3 gigabytes. To ensure this index fits in RAM, you must not only have more than that much RAM available but also must have RAM available for the rest of the *working set*. Also remember:

If you have and use multiple collections, you must consider the size of all indexes on all collections. The indexes and the working set must be able to fit in memory at the same time.

There are some limited cases where indexes do not need to fit in memory. See *Indexes that Hold Only Recent Values in RAM* (page 555).

See also:

`collStats` and `db.collection.stats()`

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit *entirely* into RAM in all cases. If the value of the indexed field increments with every insert, and most queries select recently added documents; then MongoDB only needs to keep the parts of the index that hold the most recent or "right-most" values in RAM. This allows for efficient index use for read and write operations and minimize the amount of RAM required to support the index.

Create Queries that Ensure Selectivity

Selectivity is the ability of a query to narrow results using the index. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

To ensure selectivity, write queries that limit the number of possible documents with the indexed field. Write queries that are appropriately selective relative to your indexed data.

Example

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a low-selectivity index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a *compound index* (page 489) 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 532).

For a conceptual introduction to indexes in MongoDB see *Index Concepts* (page 485).

8.4 Indexing Reference

On this page

8.4.1 Indexing Methods in the mongo Shell

Name	Description
db.collection.createIndex()	Builds an index on a collection. Use db.collection.ensureIndex().
db.collection.dropIndex()	Removes a specified index on a collection.
db.collection.dropIndexes()	Removes all indexes on a collection.
db.collection.ensureIndex()	Creates an index if it does not currently exist. If the index exists ensureIndex() does nothing.
db.collection.getIndexes()	Returns an array of documents that describe the existing indexes on a collection.
db.collection.getIndexStats()	Renders a human-readable view of the data collected by indexStats which reflects B-tree utilization.
db.collection.indexStats()	Renders a human-readable view of the data collected by indexStats which reflects B-tree utilization.
db.collection.reIndex()	Rebuilds all existing indexes on a collection.
db.collection.totalIndexSize()	Reports the total size used by the indexes on a collection. Provides a wrapper around the totalIndexSize field of the collStats output.
cursor.explain()	Reports on the query execution plan, including index use, for a cursor.
cursor.hint()	Forces MongoDB to use a specific index for a query.
cursor.max()	Specifies an exclusive upper index bound for a cursor. For use with cursor.hint().
cursor.min()	Specifies an inclusive lower index bound for a cursor. For use with cursor.hint().
cursor.snapshot()	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.

8.4.2 Indexing Database Commands

Name	Description
createIndexes	Builds one or more indexes for a collection.
dropIndexes	Removes indexes from a collection.
compact	Defragments a collection and rebuilds the indexes.
reIndex	Rebuilds all indexes on a collection.
validate	Internal command that scans for a collection's data and indexes for correctness.
indexStats	Experimental command that collects and aggregates statistics on all indexes.
geoNear	Performs a geospatial query that returns the documents closest to a given point.
geoSearch	Performs a geospatial query that uses MongoDB's *haystack index* functionality.
geoWalk	An internal command to support geospatial queries.
checkShardingIndex	Internal command that validates index on shard key.

8.4.3 Geospatial Query Selectors

Name	Description
$geoWithin	Selects geometries within a bounding *GeoJSON geometry* (page 558). The *2dsphere* (page 497) and *2d* (page 498) indexes support $geoWithin.
$geoIntersects	Selects geometries that intersect with a *GeoJSON* geometry. The *2dsphere* (page 497) index supports $geoIntersects.
$near	Returns geospatial objects in proximity to a point. Requires a geospatial index. The *2dsphere* (page 497) and *2d* (page 498) indexes support $near.
$nearSphere	Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The *2dsphere* (page 497) and *2d* (page 498) indexes support $nearSphere.

8.4.4 Indexing Query Modifiers

Name	Description
`$explain`	Forces MongoDB to report on query execution plans. See `explain()`.
`$hint`	Forces MongoDB to use a specific index. See `hint()`
`$max`	Specifies an *exclusive* upper limit for the index to use in a query. See `max()`.
`$min`	Specifies an *inclusive* lower limit for the index to use in a query. See `min()`.
`$returnKey`	Forces the cursor to only return fields included in the index.
`$snapshot`	Forces the query to use the index on the `_id` field. See `snapshot()`.

8.4.5 Other Index References

GeoJSON Objects **(page 558)** Supported GeoJSON objects.

Text Search Languages **(page 561)** Supported languages for *text indexes* (page 501) and `$text` query operations.

GeoJSON Objects

On this page

Overview

MongoDB supports the GeoJSON object types listed on this page.

To specify GeoJSON data, use a document with a `type` field specifying the GeoJSON object type and a `coordinates` field specifying the object's coordinates:

```
{ type: "<GeoJSON type>" , coordinates: <coordinates> }
```

Important: Always list coordinates in `longitude, latitude` order.

The default coordinate reference system for GeoJSON uses the *WGS84* datum.

`Point`

New in version 2.4.

The following example specifies a GeoJSON Point[10]:

[10] http://geojson.org/geojson-spec.html#point

```
{ type: "Point", coordinates: [ 40, 5 ] }
```

LineString

New in version 2.4.

The following example specifies a GeoJSON LineString[11]:

```
{ type: "LineString", coordinates: [ [ 40, 5 ], [ 41, 6 ] ] }
```

Polygon

New in version 2.4.

Polygons[12] 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 line that joins two points on a curved surface may or may not contain the same set of co-ordinates that joins those two points on a flat surface. The line that joins two points on a curved surface will be a geodesic. Carefully check points to avoid errors with shared edges, as well as overlaps and other types of intersections.

Polygons with a Single Ring The following example specifies a GeoJSON `Polygon` with an exterior ring and no interior rings (or holes). The first and last coordinates must match in order to close the polygon:

```
{
  type: "Polygon",
  coordinates: [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0  ] ] ]
}
```

For Polygons with a single ring, the ring cannot self-intersect.

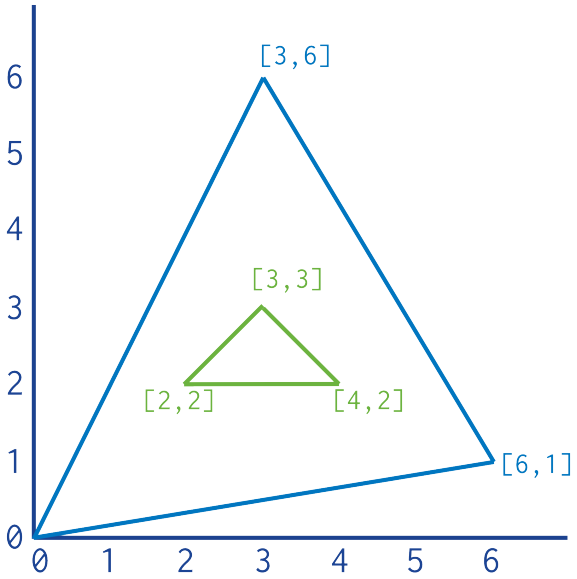
Polygons with Multiple Rings For Polygons with multiple rings:

- 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 cannot share an edge.

The following example represents a GeoJSON polygon with an interior ring:

```
{
  type : "Polygon",
  coordinates : [
     [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ],
     [ [ 2 , 2 ] , [ 3 , 3 ] , [ 4 , 2 ] , [ 2 , 2 ] ]
  ]
}
```

[11] http://geojson.org/geojson-spec.html#linestring
[12] http://geojson.org/geojson-spec.html#polygon



MultiPoint

New in version 2.6: Requires *2dsphere (Version 2)* (page 497)

The following example specifies a GeoJSON MultiPoint[13]:

```
{
  type: "MultiPoint",
  coordinates: [
     [ -73.9580, 40.8003 ],
     [ -73.9498, 40.7968 ],
     [ -73.9737, 40.7648 ],
     [ -73.9814, 40.7681 ]
  ]
}
```

MultiLineString

New in version 2.6: Requires *2dsphere (Version 2)* (page 497)

The following example specifies a GeoJSON MultiLineString[14]:

```
{
  type: "MultiLineString",
  coordinates: [
     [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],
     [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],
     [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],
     [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]
  ]
}
```

[13] http://geojson.org/geojson-spec.html#multipoint
[14] http://geojson.org/geojson-spec.html#multilinestring

MultiPolygon

New in version 2.6: Requires *2dsphere (Version 2)* (page 497)

The following example specifies a GeoJSON MultiPolygon[15]:

```
{
  type: "MultiPolygon",
  coordinates: [
     [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -73.9814, 40.7681 ],
     [ [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -73.958, 40.8003 ] ] ]
  ]
}
```

GeometryCollection

New in version 2.6: Requires *2dsphere (Version 2)* (page 497)

The following example stores coordinates of GeoJSON type GeometryCollection[16]:

```
{
  type: "GeometryCollection",
  geometries: [
     {
       type: "MultiPoint",
       coordinates: [
          [ -73.9580, 40.8003 ],
          [ -73.9498, 40.7968 ],
          [ -73.9737, 40.7648 ],
          [ -73.9814, 40.7681 ]
       ]
     },
     {
       type: "MultiLineString",
       coordinates: [
          [ [ -73.96943, 40.78519 ], [ -73.96082, 40.78095 ] ],
          [ [ -73.96415, 40.79229 ], [ -73.95544, 40.78854 ] ],
          [ [ -73.97162, 40.78205 ], [ -73.96374, 40.77715 ] ],
          [ [ -73.97880, 40.77247 ], [ -73.97036, 40.76811 ] ]
       ]
     }
  ]
}
```

Text Search Languages

The *text index* (page 501), the `$text` operator, and the `text` command [17] support the following languages:

Changed in version 2.6: MongoDB introduces version 2 of the text search feature. With version 2, text search feature supports using the two-letter language codes defined in ISO 639-1. Version 1 of text search only supported the long form of each language name.

- `da` or `danish`

[15] http://geojson.org/geojson-spec.html#multipolygon
[16] http://geojson.org/geojson-spec.html#geometrycollection
[17] The `text` command is deprecated in MongoDB 2.6.

- nl or dutch
- en or english
- fi or finnish
- fr or french
- de or german
- hu or hungarian
- it or italian
- nb or norwegian
- pt or portuguese
- ro or romanian
- ru or russian
- es or spanish
- sv or swedish
- tr or turkish

Note: If you specify a language value of "none", then the text search uses simple tokenization with no list of stop words and no stemming.

CHAPTER 9

Replication

A *replica set* in MongoDB is a group of `mongod` processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

Replication Introduction **(page 563)** An introduction to replica sets, their behavior, operation, and use.

Replication Concepts **(page 567)** The core documentation of replica set operations, configurations, architectures and behaviors.

- ***Replica Set Members*** **(page 567)** Introduces the components of replica sets.
- ***Replica Set Deployment Architectures*** **(page 575)** Introduces architectural considerations related to replica sets deployment planning.
- ***Replica Set High Availability*** **(page 583)** Presents the details of the automatic failover and recovery process with replica sets.
- ***Replica Set Read and Write Semantics*** **(page 588)** Presents the semantics for targeting read and write operations to the replica set, with an awareness of location and set configuration.

Replica Set Tutorials **(page 606)** Tutorials for common tasks related to the use and maintenance of replica sets.

Replication Reference **(page 658)** Reference for functions and operations related to replica sets.

9.1 Replication Introduction

On this page

Replication is the process of synchronizing data across multiple servers.

9.1.1 Purpose of Replication

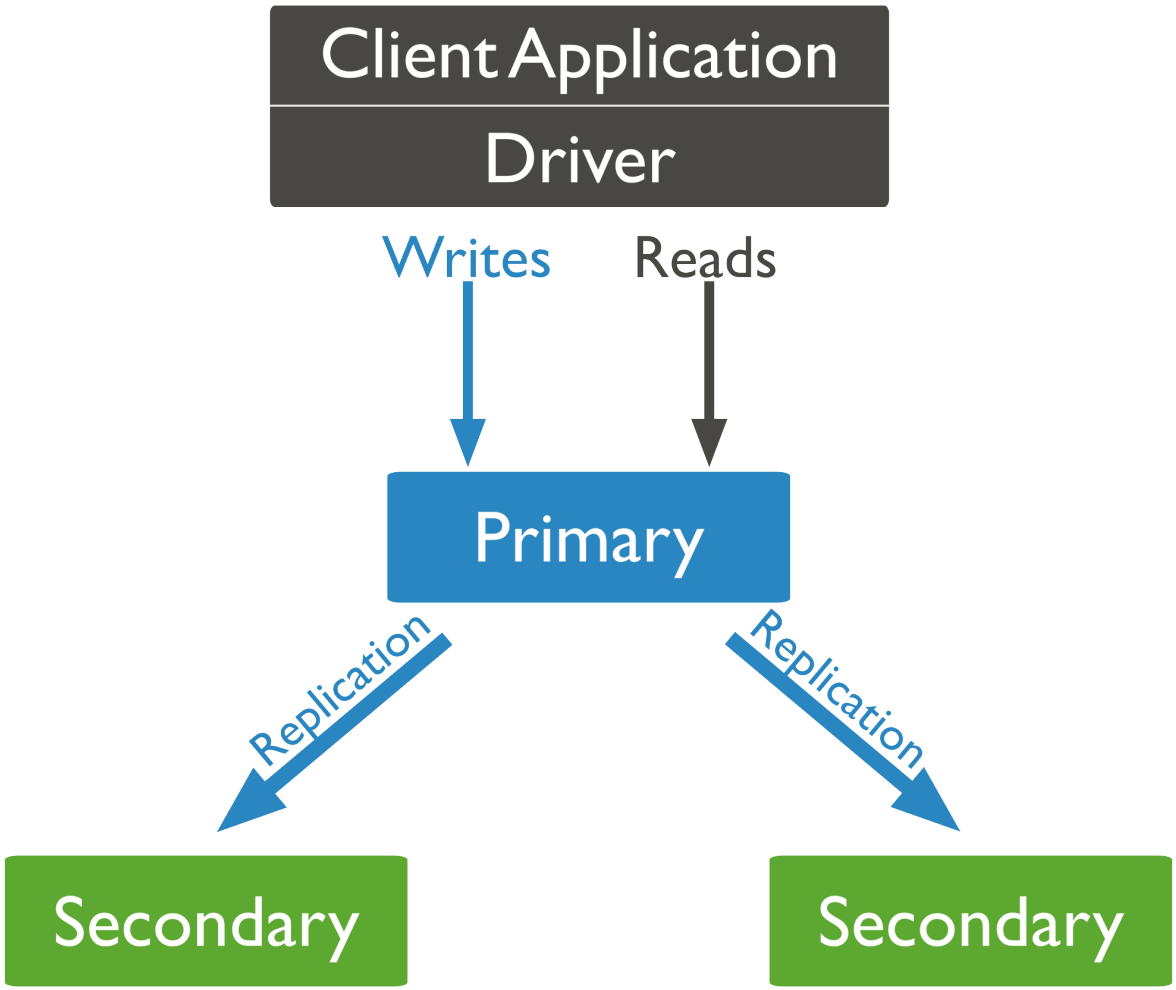
Replication provides redundancy and increases data availability. With multiple copies of data on different database servers, replication protects a database from the loss of a single server. Replication also allows you to recover from hardware failure and service interruptions. With additional copies of the data, you can dedicate one to disaster recovery, reporting, or backup.

In some cases, you can use replication to increase read capacity. Clients have the ability to send read and write operations to different servers. You can also maintain copies in different data centers to increase the locality and availability of data for distributed applications.

9.1.2 Replication in MongoDB

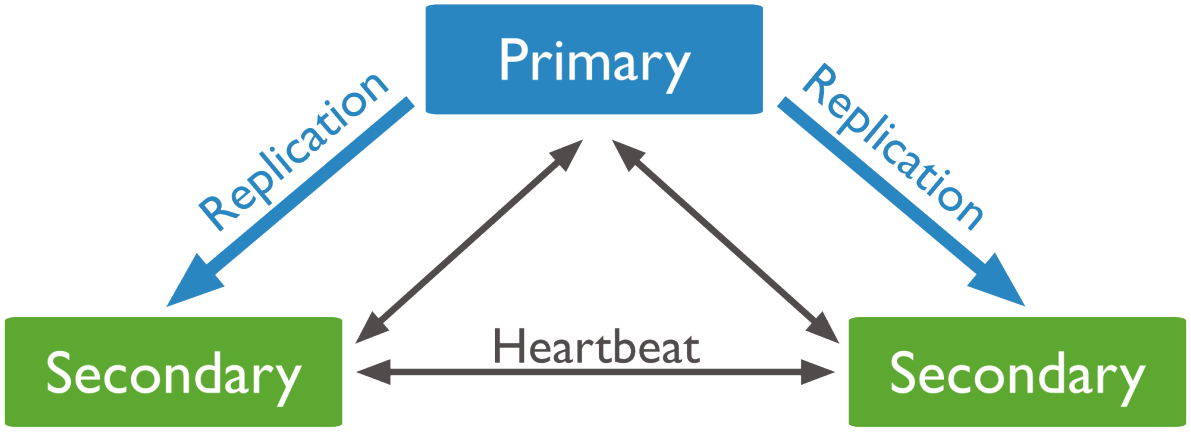
A replica set is a group of `mongod` instances that host the same data set. One `mongod`, the primary, receives all write operations. All other instances, secondaries, apply operations from the primary so that they have the same data set.

The *primary* (page 568) accepts all write operations from clients. A replica set can have only one primary. [1] To support replication, the primary records all changes to its data sets in its *oplog* (page 596). For more information on primary node operation, see *Replica Set Primary* (page 568).

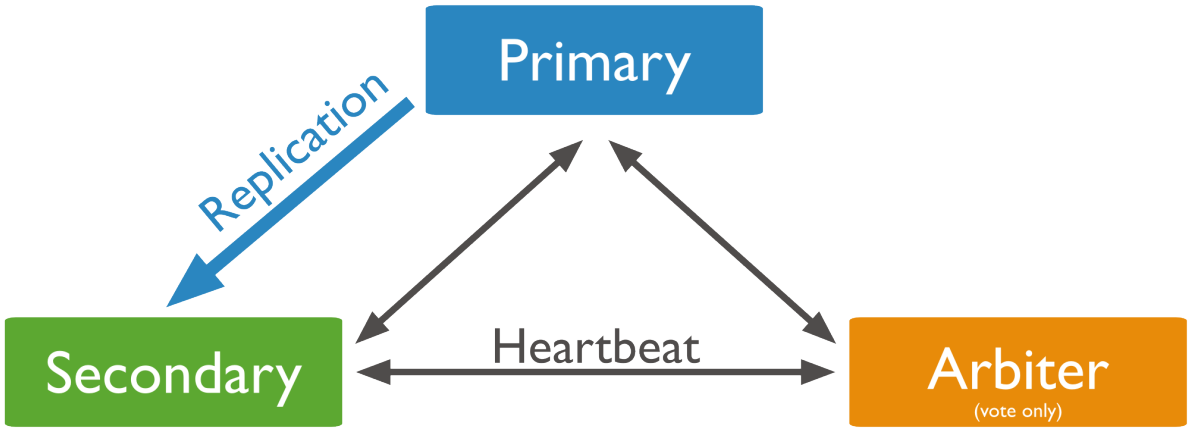


The *secondaries* (page 569) replicate the primary's oplog and apply the operations to their data sets such that the secondaries' data sets reflect the primary's data set. If the primary is unavailable, the replica set will elect a secondary to be primary. For more information on secondary members, see *Replica Set Secondary Members* (page 569).

[1] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).



You may add an extra `mongod` instance to a replica set as an *arbiter* (page 574). Arbiters do not maintain a data set. The purpose of an arbiter is to maintain a quorum in a replica set by responding to heartbeat and election requests by other replica set members. Because they do not store a data set, arbiters can be a good way to provide replica set quorum functionality with a cheaper resource cost than a fully functional replica set member with a data set. 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. For more information on arbiters, see *Replica Set Arbiter* (page 574).



An *arbiter* (page 574) will always be an arbiter whereas a *primary* (page 568) may step down and become a *secondary* (page 569) and a *secondary* (page 569) may become the primary during an election.

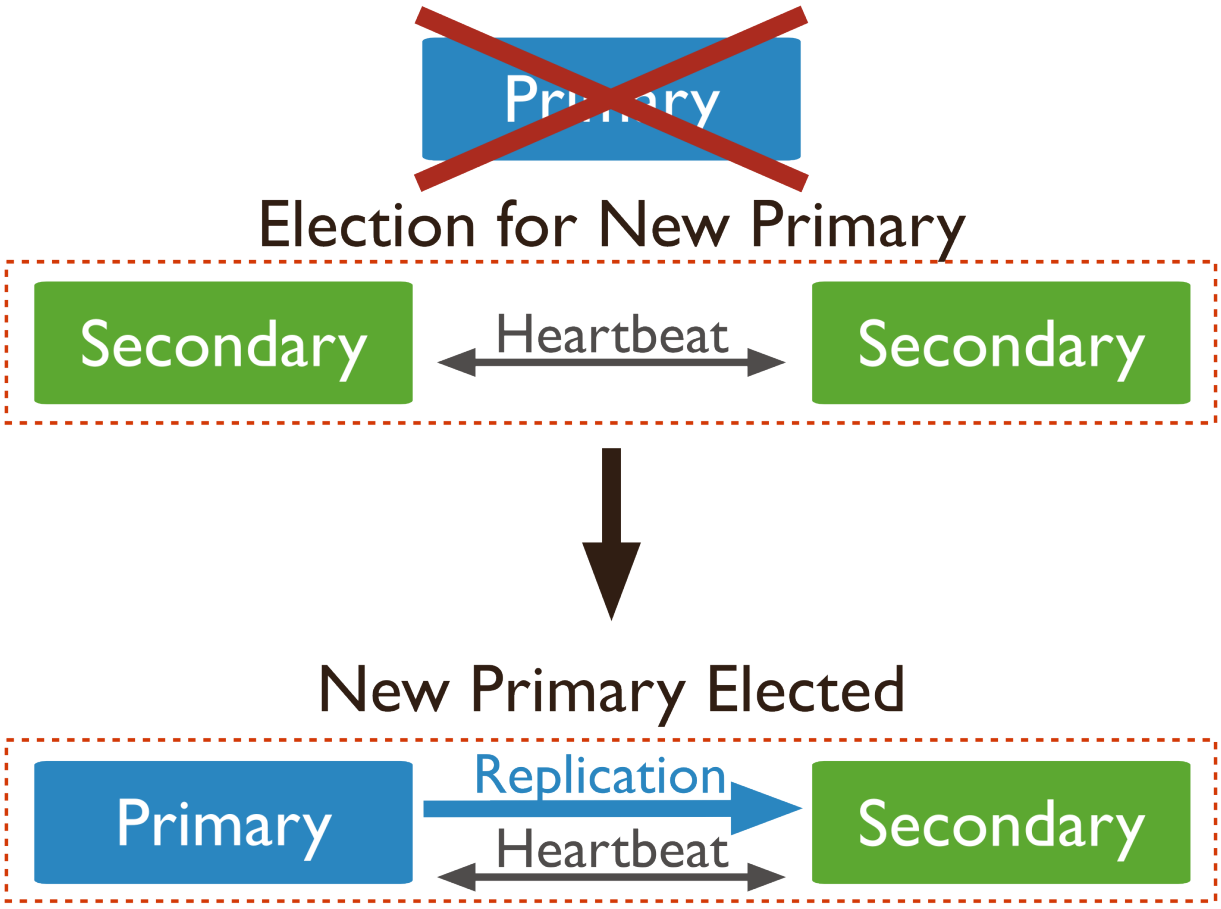
Asynchronous Replication

Secondaries apply operations from the primary asynchronously. By applying operations after the primary, sets can continue to function despite the failure of one or more members. For more information on replication mechanics, see *Replica Set Oplog* (page 596) and *Replica Set Data Synchronization* (page 598).

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 583) and *Rollbacks During Replica Set Failover* (page 587) for more information.

Read Operations

When a replica set has one and only one primary, reads from that primary provide *strict consistency*. [1]

By default, clients read from the primary; however, clients can specify a *read preference* (page 591) to send read operations to secondaries. *Asynchronous replication* (page 565) to secondaries means that reads from secondaries may return data that does not reflect the state of the data on the primary. For information on reading from replica sets, see *Read Preference* (page 591).

In MongoDB, clients can see the results of writes before they are made durable:

- Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
- Clients can read data which may be subsequently *rolled back* (page 587).

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 581), or control the outcome of elections by adjusting the `priority`

(page 662) of some members. Replica sets also support dedicated members for reporting, disaster recovery, or backup functions.

See *Priority 0 Replica Set Members* (page 570), *Hidden Replica Set Members* (page 572) and *Delayed Replica Set Members* (page 573) for more information.

9.2 Replication Concepts

These documents describe and provide examples of replica set operation, configuration, and behavior. For an overview of replication, see *Replication Introduction* (page 563). For documentation of the administration of replica sets, see *Replica Set Tutorials* (page 606). The *Replication Reference* (page 658) documents commands and operations specific to replica sets.

Replica Set Members **(page 567)** Introduces the components of replica sets.

- ***Replica Set Primary*** **(page 568)** The primary is the only member of a replica set that accepts write operations.
- ***Replica Set Secondary Members*** **(page 569)** 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 570)** Priority 0 members are secondaries that cannot become the primary.
- ***Hidden Replica Set Members*** **(page 572)** Hidden members are secondaries that are invisible to applications. These members support dedicated workloads, such as reporting or backup.
- ***Replica Set Arbiter*** **(page 574)** An arbiter does not maintain a copy of the data set but participate in elections.

Replica Set Deployment Architectures **(page 575)** Introduces architectural considerations related to replica sets deployment planning.

Replica Set High Availability **(page 583)** Presents the details of the automatic failover and recovery process with replica sets.

- ***Replica Set Elections*** **(page 583)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.
- ***Read Preference*** **(page 591)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

Replication Processes **(page 596)** Mechanics of the replication process and related topics.

Master Slave Replication **(page 600)** Master-slave replication provided redundancy in early versions of MongoDB. Replica sets replace master-slave for most use cases.

9.2.1 Replica Set Members

A *replica set* in MongoDB is a group of `mongod` processes that provide redundancy and high availability. The members of a replica set are:

Primary **(page ??).** The *primary* receives all write operations.

Secondaries **(page ??).** Secondaries replicate operations from the primary to maintain an identical data set. Secondaries may have additional configurations for special usage profiles. For example, secondaries may be *non-voting* (page 586) or *priority 0* (page 570).

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.

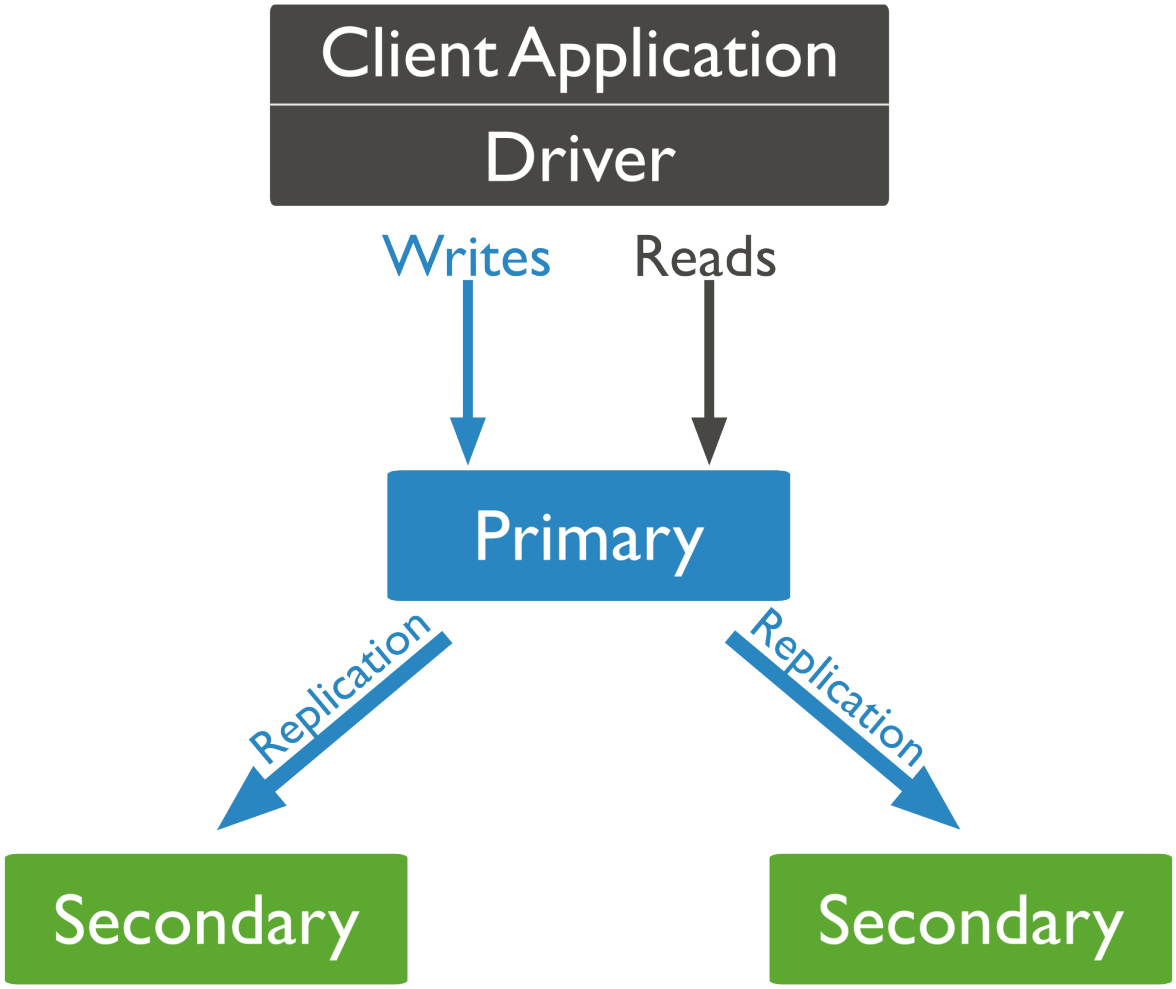
A replica set can have up to 12 members. [2] However, only 7 members can vote at a time.

The minimum requirements for a replica set are: A *primary* (page **??**), a *secondary* (page **??**), and an *arbiter* (page **??**). Most deployments, however, will keep three members that store data: A *primary* (page **??**) and two *secondary members* (page **??**).

Replica Set Primary

The primary is the only member in the replica set that receives write operations. MongoDB applies write operations on the *primary* and then records the operations on the primary's *oplog* (page 596). *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.

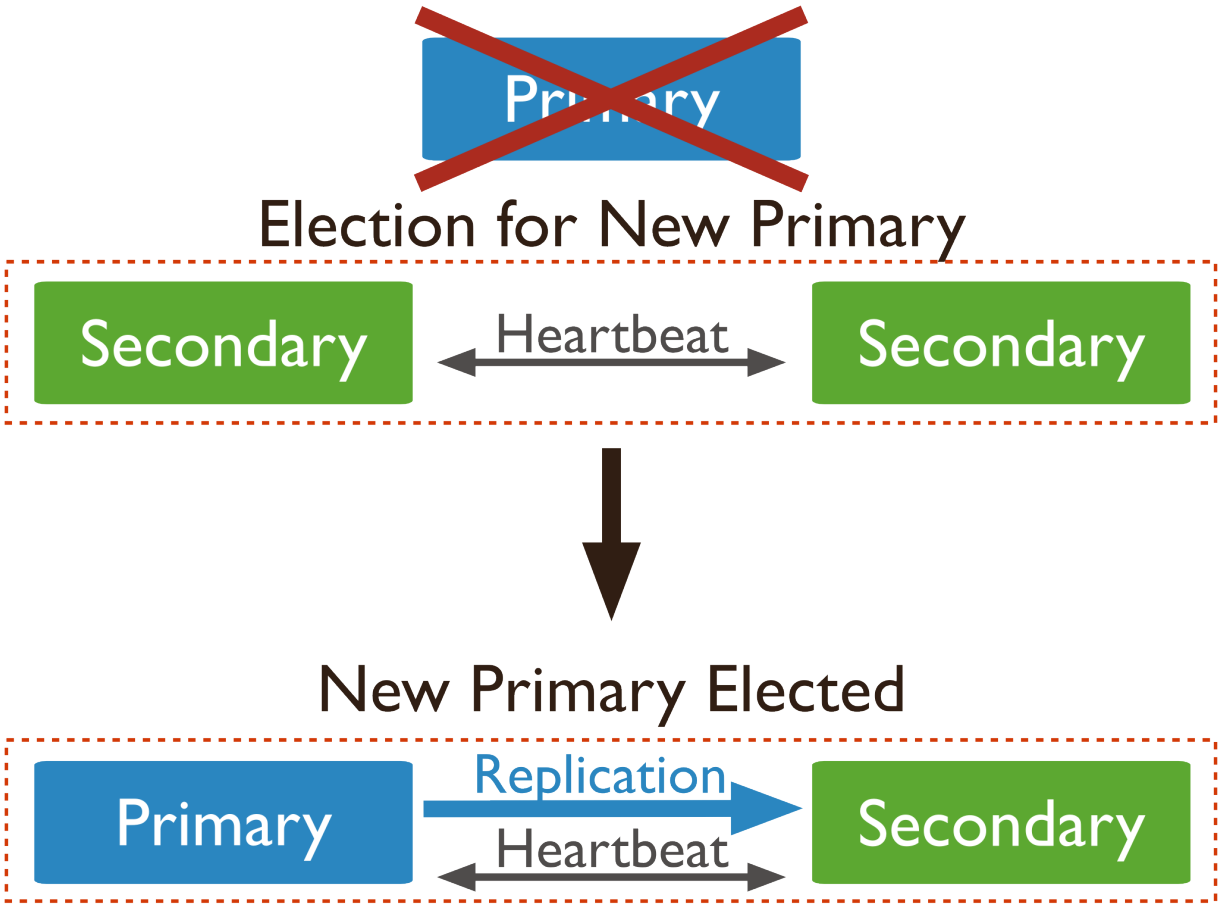


All members of the replica set can accept read operations. However, by default, an application directs its read operations to the primary member. See *Read Preference* (page 591) for details on changing the default read behavior.

[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 600) replication. Master-slave replication lacks the automatic failover capabilities.

The replica set can have at most one primary. [3] If the current primary becomes unavailable, an election determines the new primary. See *Replica Set Elections* (page 583) for more details.

In the following 3-member replica set, the primary becomes unavailable. This triggers an election which selects one of the remaining secondaries as the new primary.



Replica Set Secondary Members

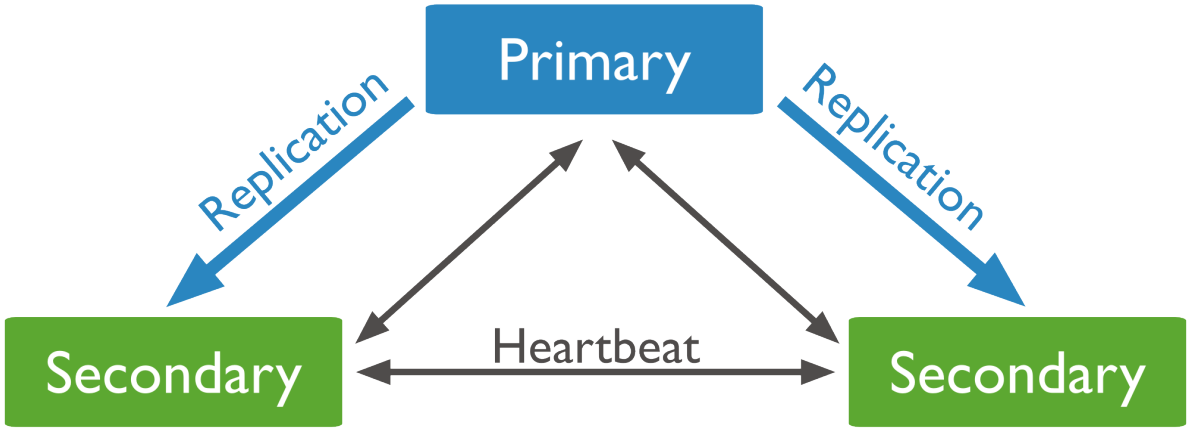
A secondary maintains a copy of the *primary's* data set. To replicate data, a secondary applies operations from the primary's *oplog* (page 596) to its own data set in an asynchronous process. A replica set can have one or more secondaries.

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 591) for more information on how clients direct read operations to replica sets.

A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an *election* to choose which of the secondaries becomes the new primary.

[3] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).



In the following three-member replica set, the primary becomes unavailable. This triggers an election where one of the remaining secondaries becomes the new primary.

See *Replica Set Elections* (page 583) 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 570).
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic. See *Hidden Replica Set Members* (page 572).
- Keep a running "historical" snapshot for use in recovery from certain errors, such as unintentionally deleted databases. See *Delayed Replica Set Members* (page 573).

Priority 0 Replica Set Members

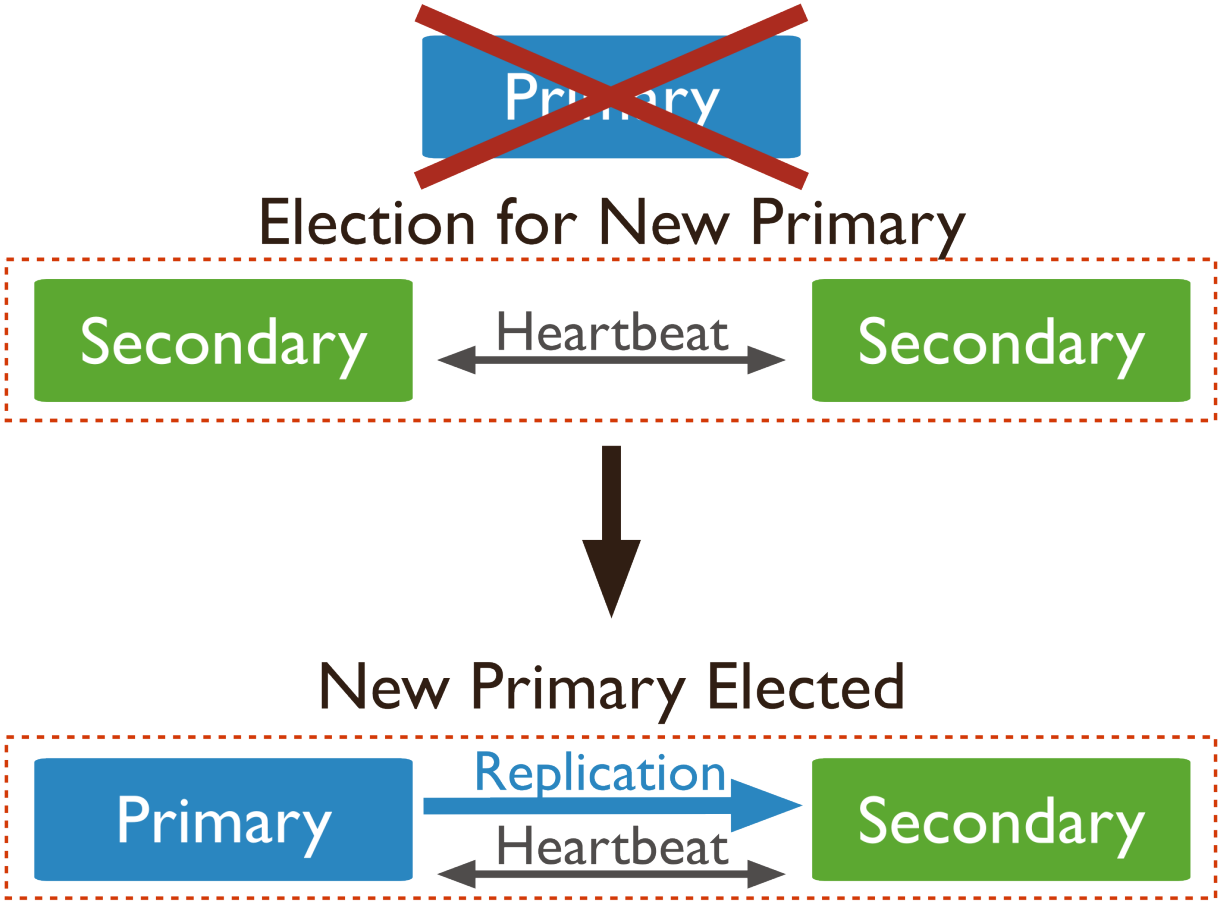
On this page

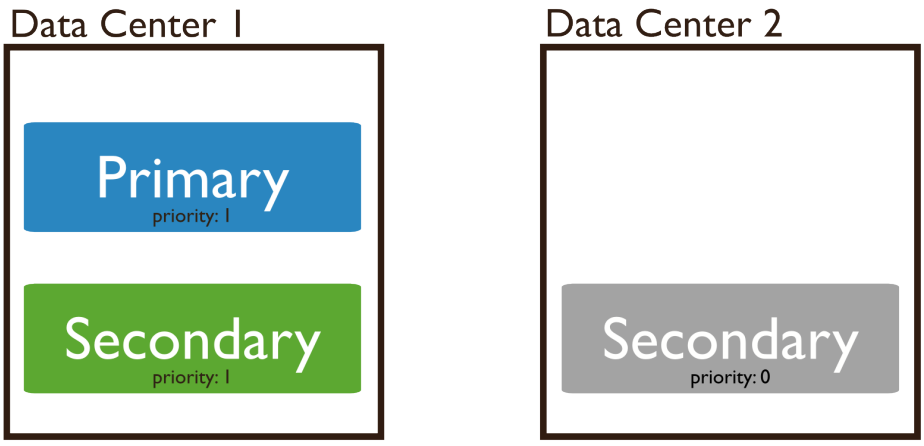
A *priority 0* member is a secondary that **cannot** become *primary*. *Priority 0* members cannot *trigger elections*. Otherwise these members function as normal secondaries. A *priority 0* member maintains a copy of the data set, accepts read operations, and votes in elections. Configure a *priority 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.

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 581), a *priority 0* standby ensures that only qualified members become primary.


Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary


Data Center 1
Primary
priority: 1
Secondary
priority: 1
Data Center 2
Secondary
priority: 0

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 572) for this purpose.

If your set already has seven voting members, also configure the member as *non-voting* (page 586).

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.

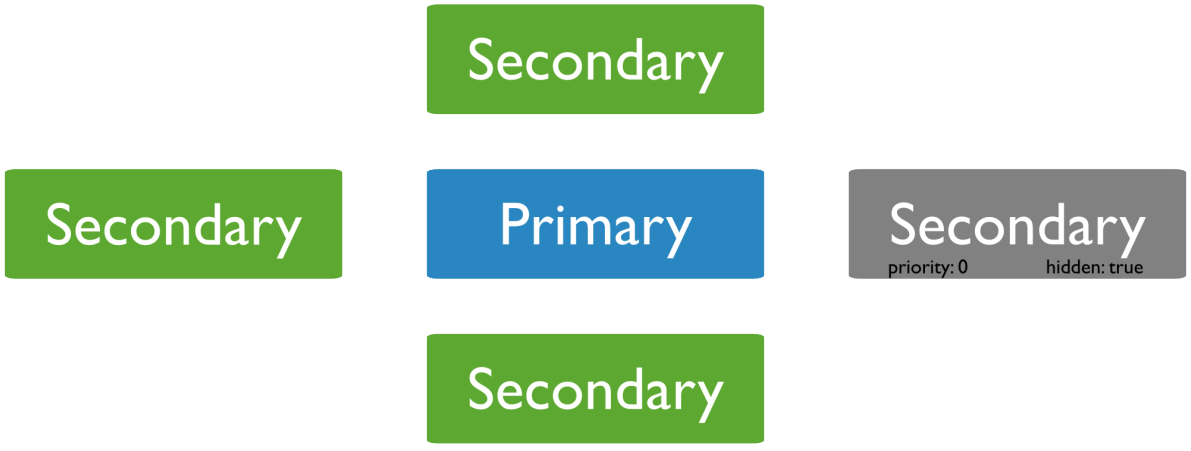
Configuration To configure a *priority 0* member, see *Prevent Secondary from Becoming Primary* (page 626).

Hidden Replica Set Members

On this page

A hidden member maintains a copy of the *primary's* data set but is **invisible** to client applications. Hidden members are good for workloads with different usage patterns from the other members in the *replica set*. Hidden members must always be *priority 0 members* (page 570) and so **cannot become primary**. The `db.isMaster()` method does not display hidden members. Hidden members, however, **may vote** in *elections* (page 583).

In the following five-member replica set, all four secondary members have copies of the primary's data set, but one of the secondary members is hidden.



Behavior

Read Operations Clients will not distribute reads with the appropriate *read preference* (page 591) to hidden members. As a result, these members receive no traffic other than basic replication. Use hidden members for dedicated tasks such as reporting and backups. *Delayed members* (page 573) should be hidden.

In a sharded cluster, `mongos` do not interact with hidden members.

Voting Hidden members *may* vote in replica set elections. If you stop a voting hidden member, ensure that the set has an active majority or the *primary* will step down.

For the purposes of backups, you can avoid stopping a hidden member with the `db.fsyncLock()` and `db.fsyncUnlock()` operations to flush all writes and lock the `mongod` instance for the duration of the backup operation.

Further Reading For more information about backing up MongoDB databases, see *MongoDB Backup Methods* (page 192). To configure a hidden member, see *Configure a Hidden Replica Set Member* (page 628).

Delayed Replica Set Members

On this page

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.

Considerations

Requirements Delayed members:

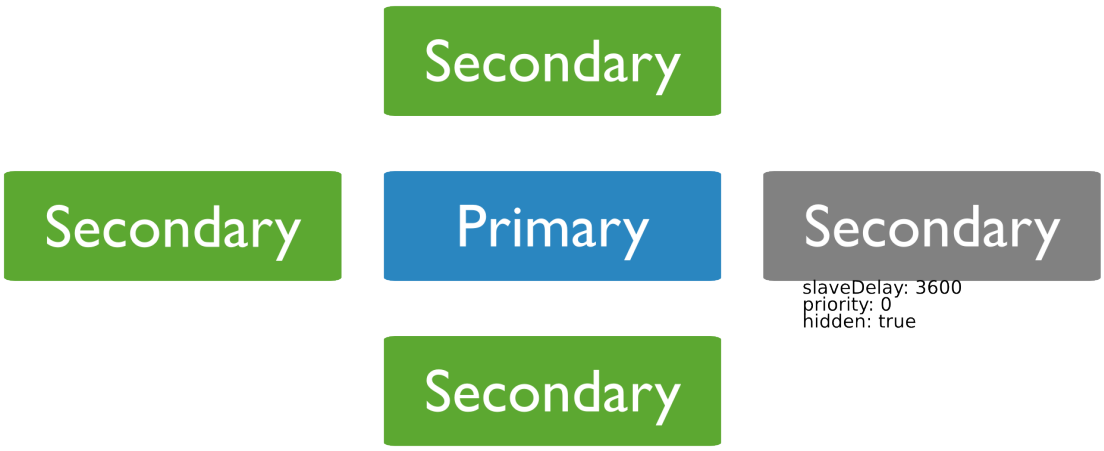
- **Must be** *priority 0* (page 570) members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** *hidden* (page 572) members. Always prevent applications from seeing and querying delayed members.
- *do* vote in *elections* for primary.

Behavior 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 597).

Sharding In sharded clusters, delayed members have limited utility when the *balancer* is enabled. Because delayed members replicate chunk migrations with a delay, the state of delayed members in a sharded cluster are not useful for recovering to a previous state of the sharded cluster if any migrations occur during the delay window.

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*.



Configuration A delayed member has its `priority` (page 662) equal to `0`, `hidden` (page 662) equal to `true`, and its `slaveDelay` (page 663) 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 629).

Replica Set Arbiter

On this page

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 583). Arbiters allow replica sets to have an uneven number of members, without the overhead of a member that replicates data.

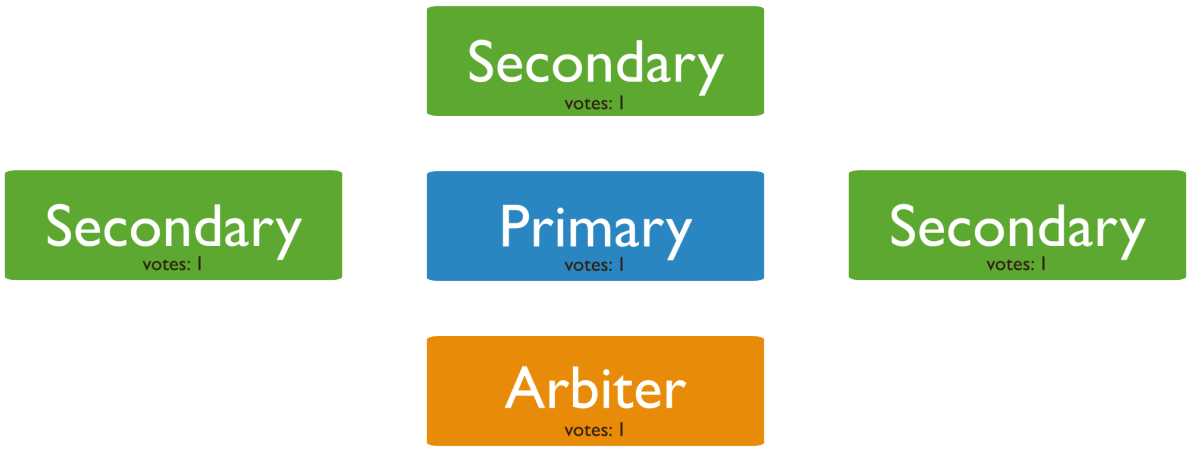
Important: Do not run an arbiter on systems that also host the primary or the secondary members of the replica set.

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 618).

Example

For example, in the following replica set, an arbiter allows the set to have an odd number of votes for elections:

Security



Authentication When running with `authorization`, 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 TLS/SSL, MongoDB will encrypt *all* communication between replica set members. See *Configure mongod and mongos for TLS/SSL* (page 338) for more information.

As with all MongoDB components, run arbiters in trusted network environments.

9.2.2 Replica Set Deployment Architectures

On this page

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.

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 572) or *delayed* (page 573) 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 662) 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 641) to ensure that operations replicate to specific data centers. Tags also support targeting read operations to specific machines.

See also:

Data Center Awareness (page 218) and *Operational Segregation in MongoDB Deployments* (page 218).

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.

Replica Set Naming

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

Deployment Patterns

The following documents describe common replica set deployment patterns. Other patterns are possible and effective depending on the application's requirements. If needed, combine features of each architecture in your own deployment:

Three Member Replica Sets **(page 577)** Three-member replica sets provide the minimum recommended architecture for a replica set.

Replica Sets with Four or More Members **(page 579)** 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 581)** Geographically distributed sets include members in multiple locations to protect against facility-specific failures, such as power outages.

Three Member Replica Sets

On this page

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.

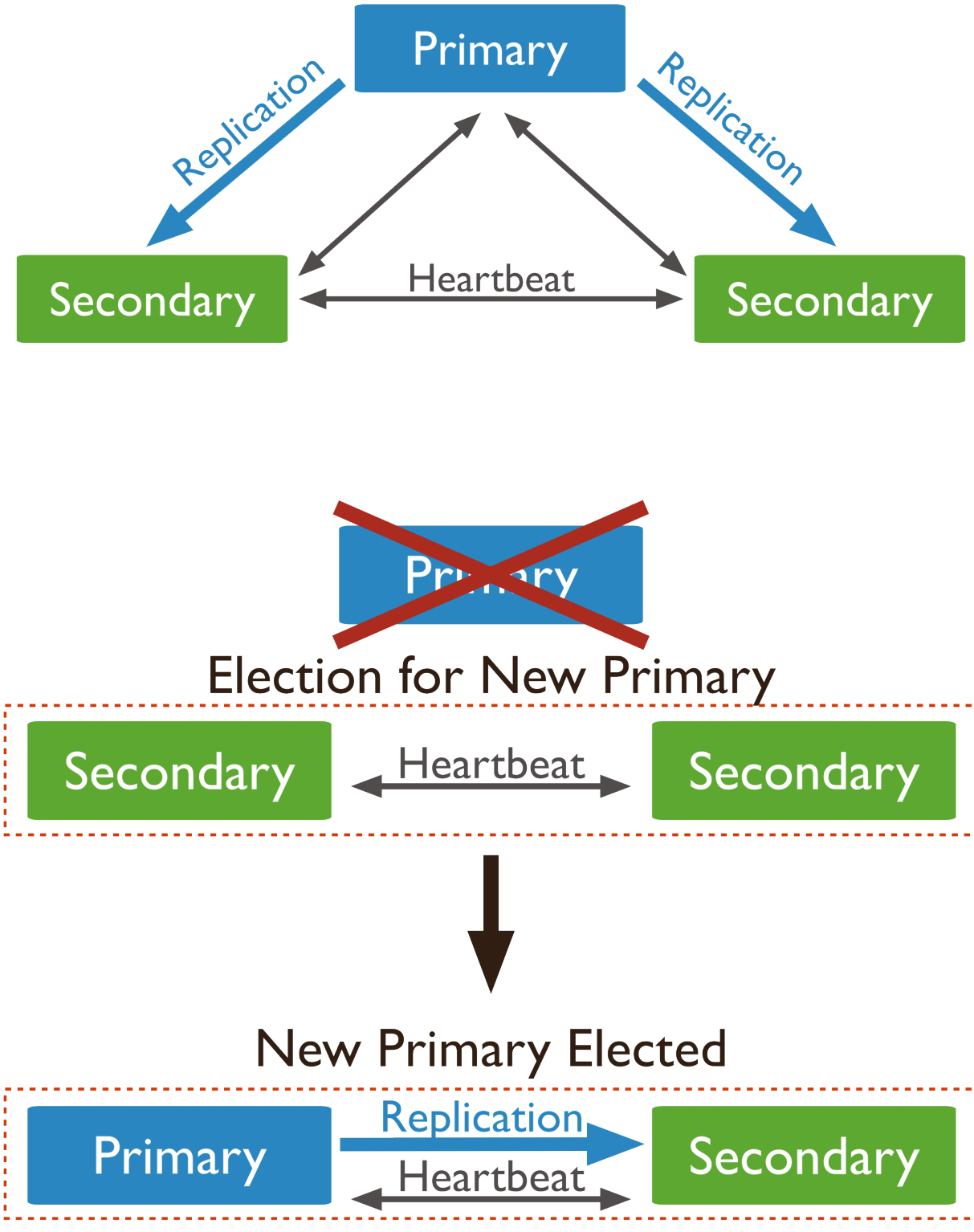
Primary with Two Secondary Members A replica set with three members that store data has:

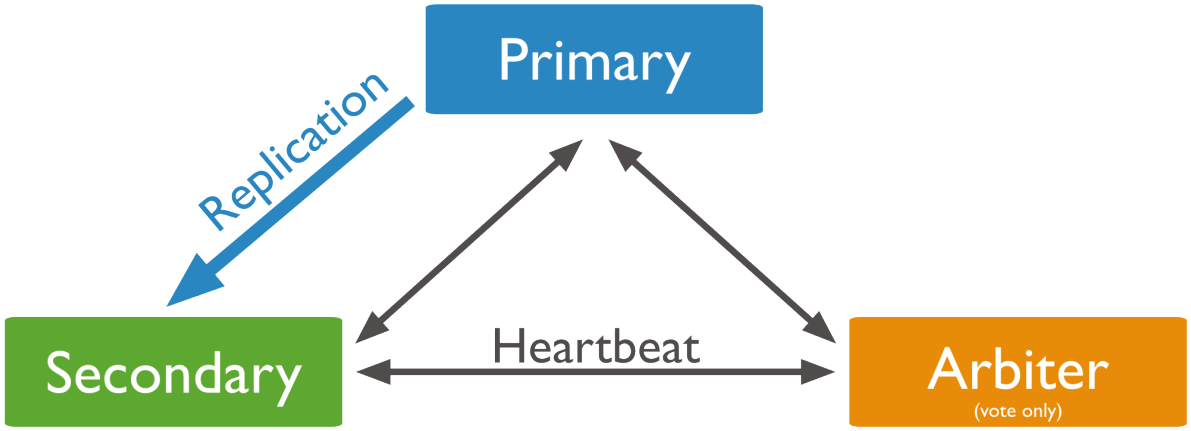
- One *primary* (page 568).
- Two *secondary* (page 569) members. Both secondaries can become the primary in an *election* (page 583).

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 583). 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 568).
- One *secondary* (page 569) member. The secondary can become primary in an *election* (page 583).
- One *arbiter* (page 574). The arbiter only votes in elections.


Primary
Replication
Replication
Secondary
Heartbeat
Secondary
Election for New Primary
Secondary
Heartbeat
Secondary
New Primary Elected
Primary
Replication
Heartbeat
Secondary



Since the arbiter does not hold a copy of the data, these deployments provides only one complete copy of the data. Arbiters require fewer resources, at the expense of more limited redundancy and fault tolerance.

However, a deployment with a primary, secondary, and an arbiter ensures that a replica set remains available if the primary *or* the secondary is unavailable. If the primary is unavailable, the replica set will elect the secondary to be primary.

See also:

Deploy a Replica Set (page 607).

Replica Sets with Four or More Members

On this page

Overview 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.

Considerations As you add new members to a replica set, consider the following:

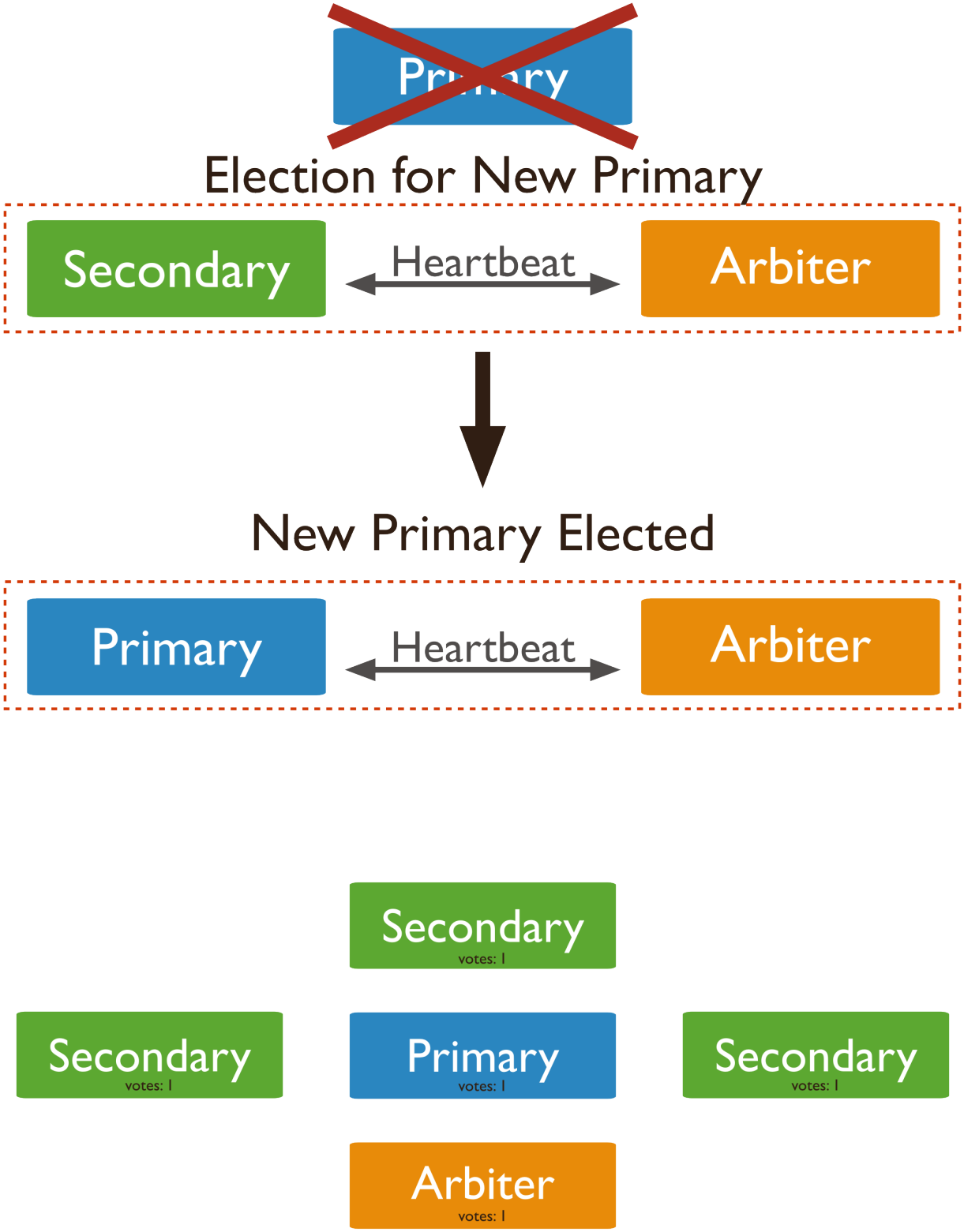
Odd Number of Voting Members Ensure that the replica set has an odd number of voting members. If you have an *even* number of voting members, deploy an *arbiter* (page **??**) so that the set has an odd number.

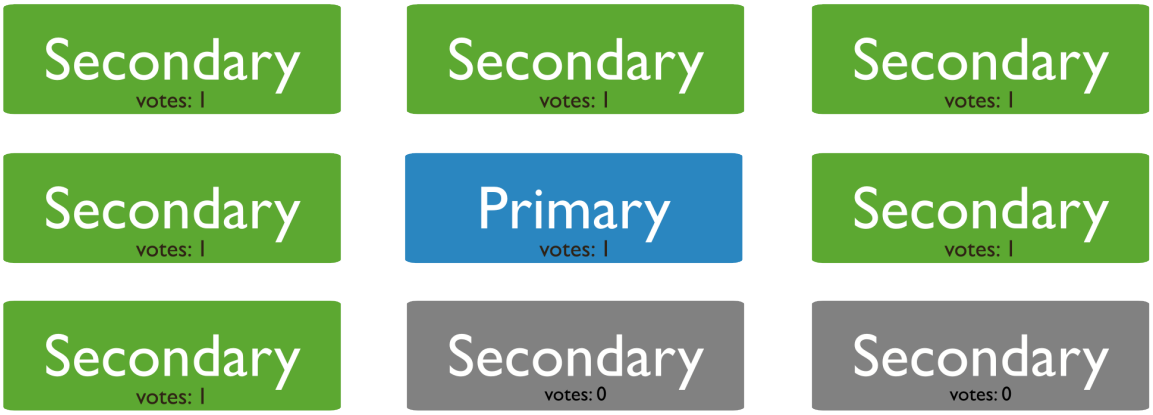
For example, the following replica set includes an arbiter to ensure an odd number of voting members.

Maximum Number of Voting Members A replica set can have up to 12 members, [4] but only 7 voting members. If the replica set already has 7 voting members, additional members must be *non-voting members* (page 586).

For example, the following 9 member replica set has 7 voting members and 2 non-voting members.

[4] 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 600) replication. Master-slave replication lacks the automatic failover capabilities.

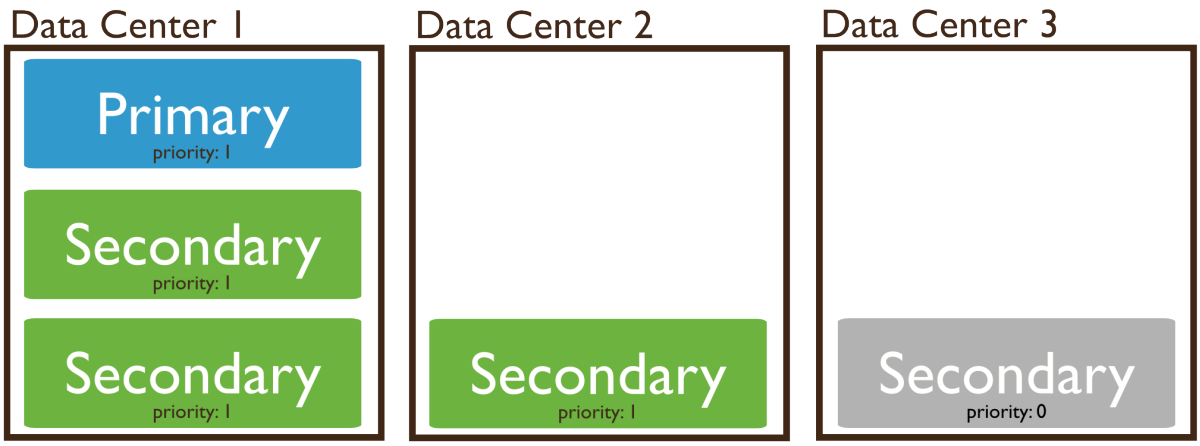

Primary
Election for New Primary
Secondary
Heartbeat
Arbiter
New Primary Elected
Primary
Heartbeat
Arbiter
Secondary
votes: 1
Secondary
votes: 1
Primary
votes: 1
Secondary
votes: 1
Arbiter
votes: 1



See *Non-Voting Members* (page 586) for more information.

Location of the Members A majority of the replica set's members should be in your application's main data center.

For example, the following 5 member replica set has the majority, 3, of its members in its main data center, `Data Center 1`.



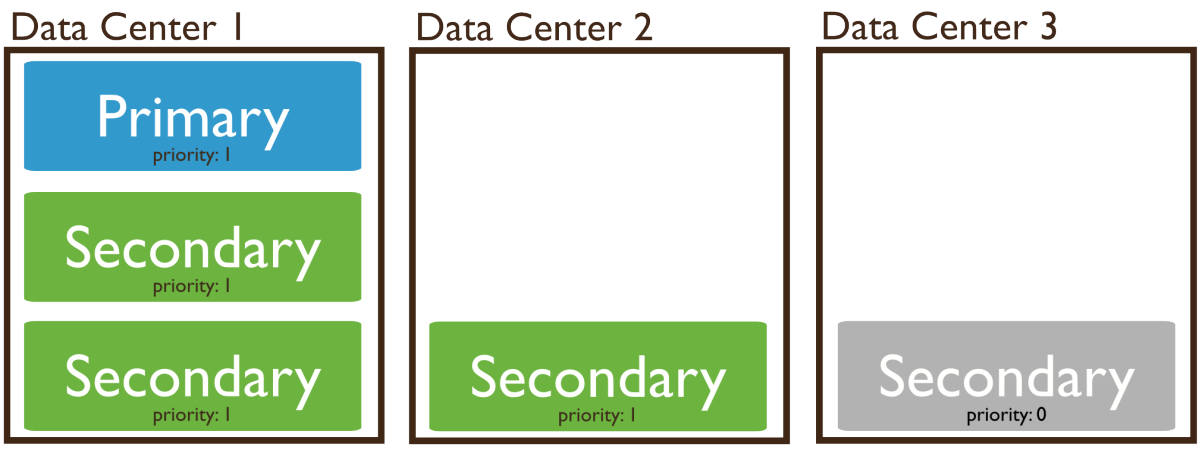
Electability of Members Some members of the replica set, such as members that have networking restraint or limited resources, should not be able to become primary in a *failover*. Configure members that should not become primary to have *priority 0* (page 570).

For example, the secondary member in the third data center with a priority of 0 cannot become primary:

See also:

Deploy a Replica Set (page 607), *Add an Arbiter to Replica Set* (page 618), and *Add Members to a Replica Set* (page 620).

Geographically Distributed Replica Sets



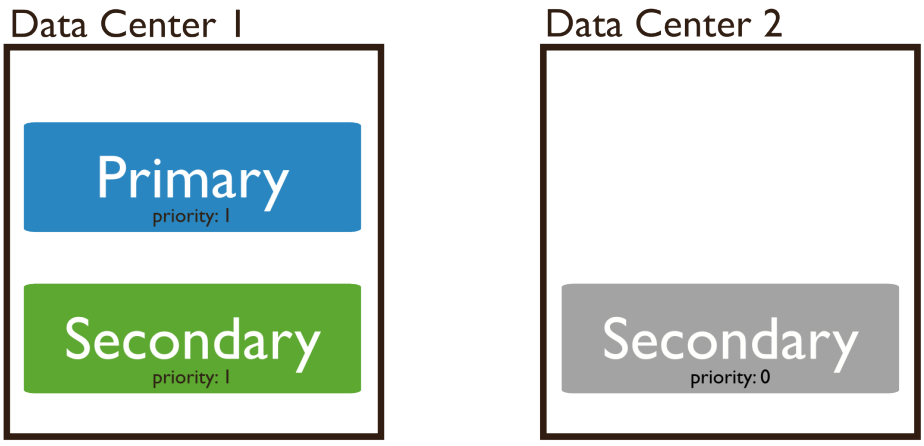
On this page

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 570) 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 570) 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 570) 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 82), 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 583).

See also:

Deploy a Geographically Redundant Replica Set (page 612).

Additional Resource MongoDB Multi-Data Center Deployments Whitepaper[5]

9.2.3 Replica Set High Availability

On this page

Replica sets provide high availability using automatic *failover*. Failover allows a *secondary* member to become *primary* if primary is unavailable. Failover, in most situations does not require manual intervention.

Replica set members keep the same data set but are otherwise independent. If the primary becomes unavailable, the replica set holds an *election* (page 583) to select a new primary. In some situations, the failover process may require a *rollback* (page 587). [6]

The deployment of a replica set affects the outcome of failover situations. To support effective failover, ensure that one facility can elect a primary if needed. Choose the facility that hosts the core application systems to host the majority of the replica set. Place a majority of voting members and all the members that can become primary in this facility. Otherwise, network partitions could prevent the set from being able to form a majority.

Failover Processes

The replica set recovers from the loss of a primary by holding an election. Consider the following:

Replica Set Elections **(page 583)** Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Rollbacks During Replica Set Failover **(page 587)** A rollback reverts write operations on a former primary when the member rejoins the replica set after a failover.

Replica Set Elections

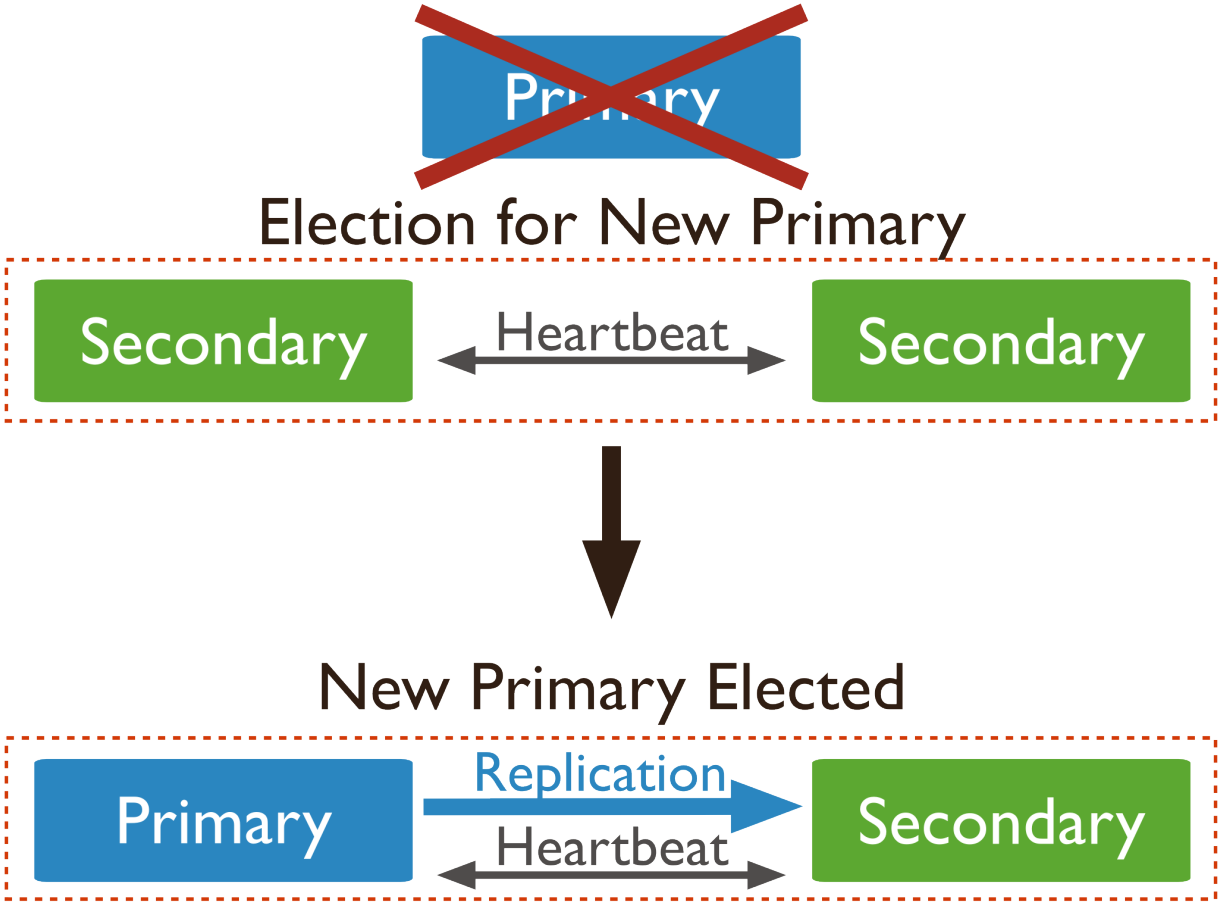
On this page

[5] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs

[6] Replica sets remove "rollback" data when needed without intervention. Administrators must apply or discard rollback data manually.

Replica sets use elections to determine which set member will become *primary*. Elections occur after initiating a replica set, and also any time the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the *failover process* (page 583).

In the following three-member replica set, the primary is unavailable. The remaining secondaries hold an election to choose a new primary.



Behavior 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 and all remaining members become read-only. MongoDB avoids elections unless necessary.

If a majority of the replica set is inaccessible or unavailable, the replica set cannot accept writes and all remaining members become read-only.

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 662) 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 570).

A replica set does *not* hold an election as long as the current primary has the highest priority value or no secondary with higher priority is within 10 seconds of the latest *oplog* entry in the set.

If a higher-priority member catches up to within 10 seconds of the latest oplog entry of the current primary, the set holds an election in order to provide the higher-priority node a chance to become primary.

Optime The `optime` is the timestamp of the last operation that a member applied from the oplog. A replica set member cannot become primary unless it has the highest (i.e. most recent) `optime` of any visible member in the set.

Connections A replica set member cannot become primary unless it can connect to a majority of the members in the replica set. For the purposes of elections, a majority refers to the total number of *votes*, rather than the total number of members.

If you have a three-member replica set, where every member has one vote, the set can elect a primary as long as two members can connect to each other. If two members are unavailable, the remaining member remains a *secondary* because it cannot connect to a majority of the set's members. If the remaining member is a *primary* and two members become unavailable, the primary steps down and becomes a secondary.

Network Partitions Network partitions affect the formation of a majority for an election. If a primary steps down and neither portion of the replica set has a majority the set will **not** elect a new primary. The replica set becomes read-only.

To avoid this situation, place a majority of instances in one data center and a minority of instances in any other data centers combined.

Election Mechanics

Election Triggering Events Replica sets hold an election any time there is no primary. Specifically, the following:

- the initiation of a new replica set.
- a secondary loses contact with a primary. Secondaries call for elections when they cannot see a primary.
- a primary steps down.

Note: *Priority 0 members* (page 570), do not trigger elections, even when they cannot connect to the primary.

A primary will step down:

- after receiving the `replSetStepDown` command.
- if one of the current secondaries is eligible for election *and* has a higher priority.
- if primary cannot contact a majority of the members of the replica set.

In some cases, modifying a replica set's configuration will trigger an election by modifying the set so that the primary must step down.

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 662) 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 662) 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 581), 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 663) setting. *Non-voting members* (page 631) have `votes` (page 663) value of `0`. All other members have `1` vote.

Note: Deprecated since version 2.6: `votes` (page 663) values greater than `1`.

Earlier versions of MongoDB allowed a member to have more than `1` vote by setting `votes` (page 663) to a value greater than `1`. Setting `votes` (page 663) to value greater than `1` now produces a warning message.

The `state` of a member also affects its eligibility to vote. Only members in the following states can vote: `PRIMARY`, `SECONDARY`, `RECOVERING`, `ARBITER`, and `ROLLBACK`.

Important: Do not alter the number of votes in a replica set to control the outcome of an election. Instead, modify the `priority` (page 662) value.

Vetoes in Elections All members of a replica set can veto an election, including *non-voting members* (page 586). 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 570) [7] is the most current member at the time of the election. In this case, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.
- If the current primary has more recent operations (i.e. a higher `optime`) than the member seeking election, from the perspective of the voting member.
- If the current primary has the same or more recent operations (i.e. a higher or equal `optime`) than the member seeking election.

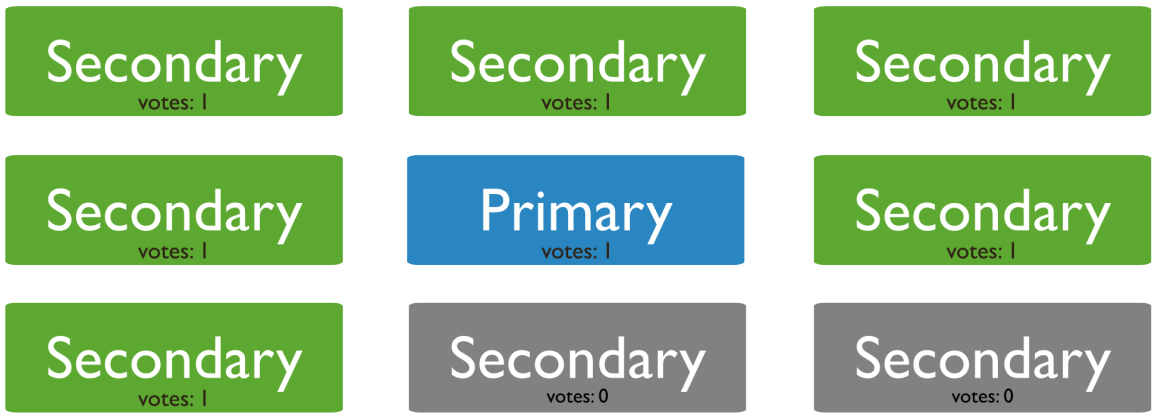
Non-Voting Members Non-voting members hold copies of the replica set's data and can accept read operations from client applications. Non-voting members do not vote in elections, but **can** *veto* (page 586) an election and become primary.

Because a replica set can have up to 12 members but only up to seven voting members, non-voting members allow a replica set to have more than seven members.

For instance, the following nine-member replica set has seven voting members and two non-voting members.

A non-voting member has a `votes` (page 663) setting equal to `0` in its member configuration:

[7] Remember that *hidden* (page 572) and *delayed* (page 573) imply *priority 0* (page 570) configuration.




```
{
  "_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 662) option. *Only* alter the number of votes in exceptional cases. For example, to permit more than seven members.

When possible, all members should have one vote. Changing the number of votes can cause the wrong members to become primary.

To configure a non-voting member, see *Configure Non-Voting Replica Set Member* (page 631).

Rollbacks During Replica Set Failover

On this page

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 and impact of the rollback.

A rollback does *not* occur if the write operations replicate to another member of the replica set before the primary steps down *and* if that member remains available and accessible to a majority of the replica set.

Collect Rollback Data When a rollback does occur, administrators must decide whether to apply or ignore the rollback data. MongoDB writes the rollback data to *BSON* files in the `rollback/` folder under the database's `dbPath` directory. The names of rollback files have the following form:

```
<database>.<collection>.<timestamp>.bson
```

For example:

```
records.accounts.2011-05-09T18-10-04.0.bson
```

Administrators must apply rollback data manually after the member completes the rollback and returns to secondary status. Use `bsondump` to read the contents of the rollback files. Then use `mongorestore` to apply the changes to the new primary.

Avoid Replica Set Rollbacks For replica sets, the default *write concern {w: 1}* (page 135) only provides acknowledgement of write operations on the primary. With the default write concern, data may be rolled back if the primary steps down before the write operations have replicated to any of the secondaries.

To prevent rollbacks of data that have been acknowledged to the client, use *{w: majority} write concern* (page 135) to guarantee that the write operations propagate to a majority of the replica set nodes before returning with acknowledgement to the issuing client.

Note:

- Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
- Clients can read data which may be subsequently *rolled back* (page 587).

Rollback Limitations A `mongod` instance will not rollback more than 300 megabytes of data. If your system must rollback more than 300 megabytes, you must manually intervene to recover the data. If this is the case, the following line will appear in your `mongod` log:

```
[replica set sync] replSet syncThread: 13410 replSet too much data to roll back
```

In this situation, save the data directly or force the member to perform an initial sync. To force initial sync, sync from a "current" member of the set by deleting the content of the `dbPath` directory for the member that requires a larger rollback.

See also:

Replica Set High Availability (page 583) and *Replica Set Elections* (page 583).

9.2.4 Replica Set Read and Write Semantics

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent.

By default, in MongoDB, read operations to a replica set return results from the *primary* (page 568).

Users may configure *read preference* on a per-connection basis to prefer that the read operations return results from the *secondary* members. If clients configure the *read preference* to permit secondary reads, read operations can return data from *secondary* members that have not replicated more recent write operations.

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. [8]

Note:

- In MongoDB, clients can see the results of writes before they are made durable:
  - Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
  - Clients can read data which may be subsequently *rolled back* (page 587).
- *Sharded clusters* where the shards are also replica sets provide the same operational semantics with regards to write and read operations.

Write Concern for Replica Sets **(page 589)** Write concern is the guarantee an application requires from MongoDB to consider a write operation successful.

Read Preference **(page 591)** Applications specify *read preference* to control how drivers direct read operations to members of the replica set.

Read Preference Processes **(page 594)** With replica sets, read operations may have additional semantics and behavior.

Write Concern for Replica Sets

On this page

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. [9]

Verify Write Operations to Replica Sets

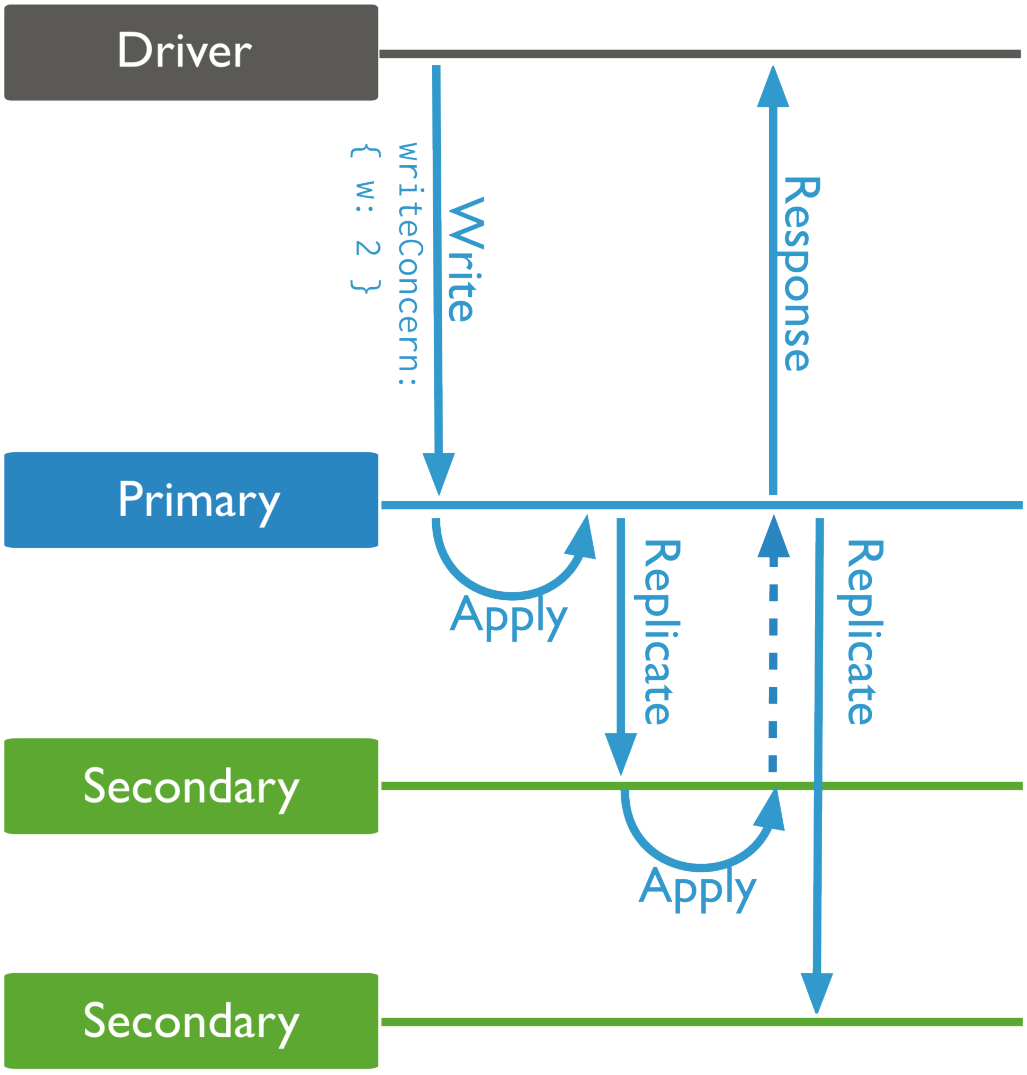
For a replica set, the default *write concern* (page 82) confirms write operations only on the primary. You can, however, override this default write concern, such as to confirm write operations on a specified number of the replica set members.

To override the default write concern, specify a write concern with each write operation. For example, the following method includes a write concern that specifies that the method return only after the write propagates to the primary and at least one secondary or the method times out after 5 seconds.

```
db.products.insert(
   { item: "envelopes", qty : 100, type: "Clasp" },
   { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

[8] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).

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


Driver
writeConcern:
{ w: 2 }
Write
Response
Primary
Apply
Replicate
Replicate
Secondary
Apply
Secondary

You can include a timeout threshold for a write concern. This prevents write operations from blocking indefinitely if the write concern is unachievable. For example, if the write concern requires acknowledgement from 4 members of the replica set and the replica set has only available 3 members, the operation blocks until those members become available. See *wtimeout* (page 136).

See also:

Write Method Acknowledgements (page 838)

Modify Default Write Concern

You can modify the default write concern for a replica set by setting the `getLastErrorDefaults` (page 664) setting in the *replica set configuration* (page 659). 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", wtimeout: 5000 }
rs.reconfig(cfg)
```

If you issue a write operation with a specific write concern, the write operation uses its own write concern instead of the default.

Note: Use of insufficient write concern can lead to *rollbacks* (page 587) in the case of *replica set failover* (page 583). Always ensure that your operations have specified the required write concern for your application.

See also:

Write Concern (page 82) and *connections-write-concern*

Custom Write Concerns

You can *tag* (page 641) the members of replica sets and use the tags to create custom write concerns. See *Configure Replica Set Tag Sets* (page 641) for information on configuring custom write concerns using tag sets.

Read Preference

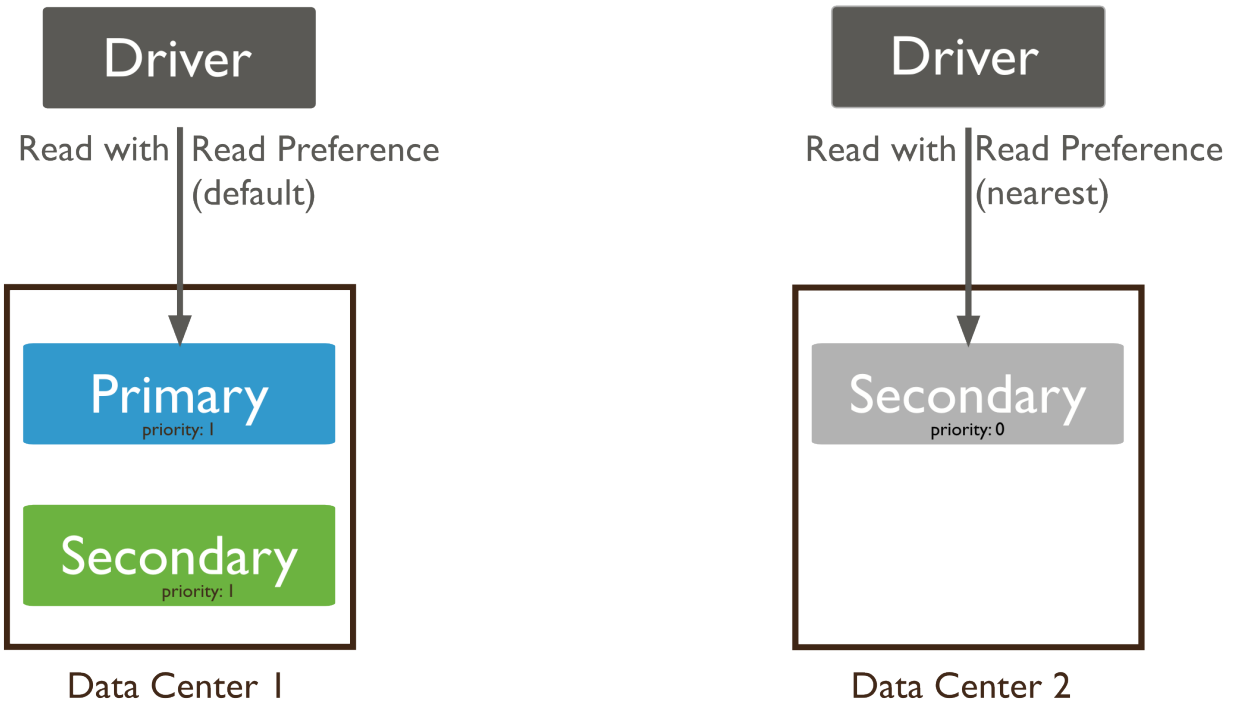
On this page

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

By default, an application directs its read operations to the *primary* member in a *replica set*. Because write operations are issued to the single primary, reading from the primary returns the latest version of a document [10].

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.

[10] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).



Important: Exercise care when specifying read preferences: Modes other than `primary` (page 670) may return stale data because with *asynchronous replication* (page 565), data in the secondary may not reflect the most recent write operations. [1]

Note: The read preference does not affect the visibility of data; i.e, clients can see the results of writes before they are made durable:

- Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
- Clients can read data which may be subsequently *rolled back* (page 587).

Use Cases

Indications The following are common use cases for using non-`primary` (page 670) read preference modes:

- Running systems operations that do not affect the front-end application.

  Note: Read preferences aren't relevant to direct connections to a single `mongod` instance. However, in order to perform read operations on a direct connection to a secondary member of a replica set, you must set a read preference, such as *secondary*.

- Providing local reads for geographically distributed applications.

  If you have application servers in multiple data centers, you may consider having a *geographically distributed replica set* (page 581) and using a non primary read preference or the `nearest` (page 671). This allows the client to read from the lowest-latency members, rather than always reading from the primary.

- Maintaining availability during a failover.

  Use `primaryPreferred` (page 670) if you want an application to read 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.

Counter-Indications In general, do *not* use `secondary` (page 670) and `secondaryPreferred` (page 671) to provide extra capacity for reads, because:

- All members of a replica have roughly equivalent write traffic, as a result secondaries will service reads at roughly the same rate as the primary.
- Because replication is asynchronous and there is some amount of delay between a successful write operation and its replication to secondaries, reading from a secondary can return out-of-date data.
- Distributing read operations to secondaries can compromise availability if *any* members of the set are unavailable because the remaining members of the set will need to be able to handle all application requests.
- For queries of sharded collections, for clusters with the *balancer* (page 698) active, secondaries may return stale results with missing or duplicated data because of incomplete or terminated migrations.

Sharding (page 675) increases read and write capacity by distributing read and write operations across a group of machines, and is often a better strategy for adding capacity.

See *Read Preference Processes* (page 594) for more information about the internal application of read preferences.

Read Preference Modes

Important: All read preference modes except `primary` (page 670) may return stale data because *secondaries* replicate operations from the primary with some delay. [1] Ensure that your application can tolerate stale data if you choose to use a non-`primary` (page 670) mode.

MongoDB `drivers` support five read preference modes.

Read Preference Mode	Description
`primary` (page 670)	Default mode. All operations read from the current replica set *primary*.
`primaryPreferred` (page 670)	In most situations, operations read from the *primary* but if it is unavailable, operations read from *secondary* members.
`secondary` (page 670)	All operations read from the *secondary* members of the replica set.
`secondaryPreferred` (page 671)	In most situations, operations read from *secondary* members but if no *secondary* members are available, operations read from the *primary*.
`nearest` (page 671)	Operations read from member of the *replica set* with the least network latency, irrespective of the member's type.

The syntax for specifying the read preference mode is specific to the driver and to the idioms of the host language[11].

Read preference modes are also available to clients connecting to a *sharded cluster* through a `mongos`. The `mongos` instance obeys specified read preferences when connecting to the *replica set* that provides each *shard* in the cluster.

In the `mongo` shell, the `readPref()` cursor method provides access to read preferences.

For more information, see *read preference background* (page 591) and *read preference behavior* (page 594). See also the documentation for your driver[12].

[11] https://api.mongodb.org/
[12] https://api.mongodb.org/

Tag Sets

Tag sets allow you to target read operations to specific members of a replica set.

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 670)
- `secondary` (page 670)
- `secondaryPreferred` (page 671)
- `nearest` (page 671)

Tags are not compatible with mode `primary` (page 670) and, in general, only apply when *selecting* (page 594) a *secondary* member of a set for a read operation. However, the `nearest` (page 671) read mode, when combined with a tag set, selects the matching member with the lowest network latency. This member may be a primary or secondary.

All interfaces use the same *member selection logic* (page 594) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For information on configuring tag sets, see the *Configure Replica Set Tag Sets* (page 641) tutorial.

For more information on how read preference *modes* (page 670) interact with tag sets, see the *documentation for each read preference mode* (page 669).

Read Preference Processes

On this page

Changed in version 2.2.

MongoDB drivers use the following procedures to direct operations to replica sets and sharded clusters. To determine how to route their operations, applications periodically update their view of the replica set's state, identifying which members are up or down, which member is *primary*, and verifying the latency to each `mongod` instance.

Member Selection

Clients, by way of their drivers, and `mongos` instances for sharded clusters, periodically update their view of the replica set's state.

When you select non-`primary` (page 670) read preference, the driver will determine which member to target using the following process:

1. Assembles a list of suitable members, taking into account member type (i.e. secondary, primary, or all members).
2. Excludes members not matching the tag sets, if specified.
3. Determines which suitable member is the closest to the client in absolute terms.

4. Builds a list of members that are within a defined ping distance (in milliseconds) of the "absolute nearest" member.

   Applications can configure the threshold used in this stage. The default "acceptable latency" is 15 milliseconds, which you can override in the drivers with their own `secondaryAcceptableLatencyMS` option. For `mongos` you can use the `--localThreshold` or `localPingThresholdMs` runtime options to set this value.

5. Selects a member from these hosts at random. The member receives the read operation.

Drivers can then associate the thread or connection with the selected member. This *request association* (page 595) is configurable by the application. See your `driver` documentation about request association configuration and default behavior.

Request Association

Important: *Request association* is configurable by the application. See your `driver` documentation about request association configuration and default behavior.

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

- The application performs a read with a different read preference,
- The thread terminates, or
- The client receives a socket exception, as is the case when there's a network error or when the `mongod` closes connections during a *failover*. This triggers a *retry* (page 595), which may be transparent to the application.

When using request association, if the client detects that the set has elected a new *primary*, the driver will discard all associations between threads and members.

Auto-Retry

Connections between MongoDB drivers and `mongod` instances in a *replica set* must balance two concerns:

1. The client should attempt to prefer current results, and any connection should read from the same member of the replica set as much as possible. Requests should prefer *request association* (page 595) (e.g. *pinning*).
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:

- Reuse a connection to a specific `mongod` for as long as possible after establishing a connection to that instance. This connection is *pinned* to this `mongod`.
- Attempt to reconnect to a new member, obeying existing *read preference modes* (page 670), if the connection to `mongod` is lost.

  Reconnections are transparent to the application itself. If the connection permits reads from *secondary* members, after reconnecting, the application can receive two sequential reads returning from different secondaries. Depending on the state of the individual secondary member's replication, the documents can reflect the state of your database at different moments.

- Return an error *only* after attempting to connect to three members of the set that match the *read preference mode* (page 670) and *tag set* (page 594). 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 670).

- After detecting a failover situation, [13] the driver attempts to refresh the state of the replica set as quickly as possible.

Read Preference in Sharded Clusters

Changed in version 2.2: Before version 2.2, `mongos` did not support the *read preference mode semantics* (page 670).

In most *sharded clusters*, each shard consists of a *replica set*. As such, read preferences are also applicable. With regard to read preference, read operations in a sharded cluster are identical to unsharded replica sets.

Unlike simple replica sets, in sharded clusters, all interactions with the shards pass from the clients to the `mongos` instances that are actually connected to the set members. `mongos` is then responsible for the application of read preferences, which is transparent to applications.

There are no configuration changes required for full support of read preference modes in sharded environments, as long as the `mongos` is at least version 2.2. All `mongos` maintain their own connection pool to the replica set members. As a result:

- A request without a specified preference has `primary` (page 670), the default, unless, the `mongos` reuses an existing connection that has a different mode set.

  To prevent confusion, always explicitly set your read preference mode.

- All `nearest` (page 671) and latency calculations reflect the connection between the `mongos` and the `mongod` instances, not the client and the `mongod` instances.

  This produces the desired result, because all results must pass through the `mongos` before returning to the client.

9.2.5 Replication Processes

Members of a *replica set* replicate data continuously. First, a member uses *initial sync* to capture the data set. Then the member continuously records and applies every operation that modifies the data set. Every member records operations in its *oplog* (page 596), which is a *capped collection*.

Replica Set Oplog **(page 596)** The oplog records all operations that modify the data in the replica set.

Replica Set Data Synchronization **(page 598)** Secondaries must replicate all changes accepted by the primary. This process is the basis of replica set operations.

Replica Set Oplog

[13] 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*.

On this page

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, in the `local.oplog.rs` (page 666) collection, which allows 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 too stale to continue replicating. However, most replica sets have much lower operation volumes, and their oplogs can hold much higher numbers of operations.

Before `mongod` creates an oplog, you can specify its size with the `oplogSizeMB` 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 634) 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, but will always allocate at least 1 gigabyte and never more than 50 gigabytes.
- 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 with a minimal amount of write operations, a smaller 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 `rs.printReplicationInfo()` method. For more information on oplog status, see *Check the Size of the Oplog* (page 657).

Under various exceptional situations, updates to a *secondary's* oplog might lag behind the desired performance time. Use `db.getReplicationInfo()` from a secondary member and the `replication status` output to assess the current state of replication and determine if there is any unintended replication delay.

See *Replication Lag* (page 654) for more information.

Replica Set Data Synchronization

On this page

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 598) to populate new members with the full data set, and replication to apply ongoing changes to the entire data set.

Initial Sync

Initial sync copies all the data from one member of the replica set to another member. A member uses initial sync when the member has no data, such as when the member is new, or when the member has data but is missing a history of the set's replication.

When you perform an initial sync, MongoDB does the following:

1. Clones all databases. To clone, the `mongod` queries every collection in each source database and inserts all data into its own copies of these collections. At this time, _id indexes are also built.
2. Applies all changes to the data set. Using the oplog from the source, the `mongod` updates its data set to reflect the current state of the replica set.
3. Builds all indexes on all collections (except _id indexes, which were already completed).

   When the `mongod` finishes building all index builds, the member can transition to a normal state, i.e. *secondary*.

To perform an initial sync, see *Resync a Member of a Replica Set* (page 640).

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, both members must have the same value for the `buildIndexes` (page 661) setting.

Beginning in version 2.2, secondaries avoid syncing from *delayed members* (page 573) and *hidden members* (page 572).

Validity and Durability

In a replica set, the set can have at most one primary and only the primary can accept write operations. [14] Secondaries apply operations from the primary asynchronously to provide *eventual consistency*.

Journaling provides single-instance write durability. Without journaling, if a MongoDB instance terminates ungracefully, you must assume that the database is in an invalid state.

In MongoDB, clients can see the results of writes before they are made durable:

- Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
- Clients can read data which may be subsequently *rolled back* (page 587).

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 481).

Optionally, you can disable all pre-fetching or only pre-fetch the index on the `_id` field. See the `secondaryIndexPrefetch` setting for more information.

[14] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).

9.2.6 Master Slave Replication

On this page

Important: *Replica sets* (page 567) replace *master-slave* replication for most use cases. If possible, use replica sets rather than master-slave replication for all new production deployments. This documentation remains to support legacy deployments and for archival purposes only.

In addition to providing all the functionality of master-slave deployments, replica sets are also more robust for production use. Master-slave replication preceded replica sets and made it possible to have a large number of non-master (i.e. slave) nodes, as well as to restrict replicated operations to only a single database; however, master-slave replication provides less redundancy and does not automate failover. See *Deploy Master-Slave Equivalent using Replica Sets* (page 602) 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 602).

Fundamental Operations

Initial Deployment

To configure a *master-slave* deployment, start two `mongod` instances: one in master mode, and the other in slave mode.

To start a `mongod` instance in master mode, invoke `mongod` as follows:

```
mongod --master --dbpath /data/masterdb/
```

With the *`--master`* option, the `mongod` will create a `local.oplog.$main` (page 666) collection, which the "operation log" that queues operations that the slaves will apply to replicate operations from the master. The *`--dbpath`* is optional.

To start a `mongod` instance in slave mode, invoke `mongod` as follows:

```
mongod --slave --source <masterhostname><:<port>> --dbpath /data/slavedb/
```

Specify the hostname and port of the master instance to the *`--source`* argument. The *`--dbpath`* is optional.

For slave instances, MongoDB stores data about the source server in the `local.sources` (page 667) 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 667) specifying the master instance, as in the following operation in the `mongo` shell:

```
use local
db.sources.find()
db.sources.insert( { host: <masterhostname> <,only: databasename> } );
```

In line 1, you switch context to the `local` database. In line 2, the `find()` operation should return no documents, to ensure that there are no documents in the `sources` collection. Finally, line 3 uses `db.collection.insert()`

to insert the source document into the local.sources (page 667) collection. The model of the local.sources (page 667) document is as follows:

host
The host field specifies the mastermongod instance, and holds a resolvable hostname, i.e. IP address, or a name from a host file, or preferably a fully qualified domain name.

You can append <:port> to the host name if the mongod is not running on the default 27017 port.

only
Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

Operational Considerations for Replication with Master Slave Deployments

Master instances store operations in an *oplog* which is a *capped collection* (page 219). As a result, if a slave falls too far behind the state of the master, it cannot "catchup" and must re-sync from scratch. Slave may become out of sync with a master if:

- The slave falls far behind the data updates available from that master.
- The slave stops (i.e. shuts down) and restarts later after the master has overwritten the relevant operations from the master.

When slaves are out of sync, replication stops. Administrators must intervene manually to restart replication. Use the resync command. Alternatively, the *--autoresync* allows a slave to restart replication automatically, after ten second pause, when the slave falls out of sync with the master. With *--autoresync* specified, the slave will only attempt to re-sync once in a ten minute period.

To prevent these situations you should specify a larger oplog when you start the master instance, by adding the *--oplogSize* option when starting mongod. If you do not specify *--oplogSize*, mongod will allocate 5% of available disk space on start up to the oplog, with a minimum of 1GB for 64bit machines and 50MB for 32bit machines.

Run time Master-Slave Configuration

MongoDB provides a number of command line options for mongod instances in *master-slave* deployments. See the *Master-Slave Replication Command Line Options* for options.

Diagnostics

On a *master* instance, issue the following operation in the mongo shell to return replication status from the perspective of the master:

```
rs.printReplicationInfo()
```

New in version 2.6: rs.printReplicationInfo(). For previous versions, use db.printReplicationInfo().

On a *slave* instance, use the following operation in the mongo shell to return the replication status from the perspective of the slave:

```
rs.printSlaveReplicationInfo()
```

New in version 2.6: rs.printSlaveReplicationInfo(). For previous versions, use db.printSlaveReplicationInfo().

Use the serverStatus as in the following operation, to return status of the replication:

```
db.serverStatus()
```

See *server status repl fields* for documentation of the relevant section of output.

Security

When running with `authorization` enabled, in *master-slave* deployments configure a `keyFile` so that slave `mongod` instances can authenticate and communicate with the master `mongod` instance.

To enable authentication and configure the `keyFile` add the following option to your configuration file:

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

Note: You may chose to set these run-time configuration options using the *`--keyFile`* option on the command line.

Setting `keyFile` enables authentication and specifies a key file for the `mongod` instances to use when authenticating to each other. The content of the key file is arbitrary but must be the same on all members of the deployment can connect to each other.

The key file must be less one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or "world" permissions on UNIX systems. Use the following command to use the OpenSSL package to generate "random" content for use in a key file:

```
openssl rand -base64 741
```

See also:

Security (page 313) for more information about security in MongoDB

Ongoing Administration and Operation of Master-Slave Deployments

Deploy Master-Slave Equivalent using Replica Sets

If you want a replication configuration that resembles *master-slave* replication, using *replica sets* replica sets, consider the following replica configuration document. In this deployment hosts `<master>` and `<slave>` [15] 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 659) for more information about replica set configurations.

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 directories from previous secondaries and add them as new secondaries to the new replica set.

1. To confirm that the current instance is master, run:

[15] In replica set configurations, the `host` (page 661) field must hold a resolvable hostname.

```
db.isMaster()
```

This should return a document that resembles the following:

```
{
        "ismaster" : true,
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "localTime" : ISODate("2013-07-08T20:15:13.664Z"),
        "ok" : 1
}
```

2. Shut down the `mongod` processes on the master and all slave(s), using the following command while connected to each instance:

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

3. Back up your `/data/db` directories, in case you need to revert to the master-slave deployment.
4. Start the former master with the *`--replSet`* option, as in the following:

   ```
   mongod --replSet <setname>
   ```

5. Connect to the `mongod` with the `mongo` shell, and initiate the replica set with the following command:

   ```
   rs.initiate()
   ```

   When the command returns, you will have successfully deployed a one-member replica set. You can check the status of your replica set at any time by running the following command:

   ```
   rs.status()
   ```

You can now follow the *convert a standalone to a replica set* (page 619) tutorial to deploy your replica set, picking up from the *Expand the Replica Set* (page 620) section.

Failing over to a Slave (Promotion)

To permanently failover from a unavailable or damaged *master* (`A` in the following example) to a *slave* (`B`):

1. Shut down `A`.
2. Stop `mongod` on `B`.
3. Back up and move all data files that begin with `local` on `B` from the `dbPath`.

   Warning: Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

4. Restart `mongod` on `B` with the *`--master`* option.

Note: This is a one time operation, and is not reversible. `A` cannot become a slave of `B` until it completes a full resync.

Inverting Master and Slave

If you have a *master* (`A`) and a *slave* (`B`) and you would like to reverse their roles, follow this procedure. The procedure assumes `A` is healthy, up-to-date and available.

If A is not healthy but the hardware is okay (power outage, server crash, etc.), skip steps 1 and 2 and in step 8 replace all of A's files with B's files in step 8.

If A is not healthy and the hardware is not okay, replace A with a new machine. Also follow the instructions in the previous paragraph.

To invert the master and slave in a deployment:

1. Halt writes on A using the *fsync* command.
2. Make sure B is up to date with the state of A.
3. Shut down B.
4. Back up and move all data files that begin with local on B from the dbPath to remove the existing local.sources data.

   Warning: Removing local.* is irrevocable and cannot be undone. Perform this step with extreme caution.

5. Start B with the *--master* option.
6. Do a write on B, which primes the *oplog* to provide a new sync start point.
7. Shut down B. B will now have a new set of data files that start with local.
8. Shut down A and replace all files in the dbPath of A that start with local with a copy of the files in the dbPath of B that begin with local.

   Considering compressing the local files from B while you copy them, as they may be quite large.

9. Start B with the *--master* option.
10. Start A with all the usual slave options, but include *fastsync*.

Creating a Slave from an Existing Master's Disk Image

If you can stop write operations to the *master* for an indefinite period, you can copy the data files from the master to the new *slave* and then start the slave with *--fastsync*.

Warning: Be careful with *--fastsync*. If the data on both instances is **not** identical, a discrepancy will exist forever.

fastsync is a way to start a slave by starting with an existing master disk image/backup. This option declares that the administrator guarantees the image is correct and completely up-to-date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a Slave from an Existing Slave's Disk Image

You can just copy the other *slave's* data file snapshot without any special options. Only take data snapshots when a mongod process is down or locked using db.fsyncLock().

Resyncing a Slave that is too Stale to Recover

Slaves asynchronously apply write operations from the *master* that the slaves poll from the master's *oplog*. The oplog is finite in length, and if a slave is too far behind, a full resync will be necessary. To resync the slave, connect to a slave using the `mongo` and issue the `resync` command:

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

This forces a full resync of all data (which will be very slow on a large database). You can achieve the same effect by stopping `mongod` on the slave, deleting the entire content of the `dbPath` on the slave, and restarting the `mongod`.

Slave Chaining

Slaves cannot be "chained." They must all connect to the *master* directly.

If a slave attempts "slave from" another slave you will see the following line in the `mongod` 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 667) collection.

Example

Consider the following: If you accidentally set an incorrect hostname for the slave's *source*, as in the following example:

```
mongod --slave --source prod.mississippi
```

You can correct this, by restarting the slave without the *--slave* and *--source* arguments:

```
mongod
```

Connect to this `mongod` instance using the `mongo` shell and update the `local.sources` (page 667) 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 667) 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*.

9.3 Replica Set Tutorials

The administration of *replica sets* includes the initial deployment of the set, adding and removing members to a set, and configuring the operational parameters and properties of the set. Administrators generally need not intervene in failover or replication processes as MongoDB automates these functions. In the exceptional situations that require manual interventions, the tutorials in these sections describe processes such as resyncing a member. The tutorials in this section form the basis for all replica set administration.

Replica Set Deployment Tutorials **(page 606)** Instructions for deploying replica sets, as well as adding and removing members from an existing replica set.

- ***Deploy a Replica Set*** **(page 607)** Configure a three-member replica set for production systems.
- ***Convert a Standalone to a Replica Set*** **(page 619)** Convert an existing standalone `mongod` instance into a three-member replica set.
- ***Add Members to a Replica Set*** **(page 620)** Add a new member to an existing replica set.
- ***Remove Members from Replica Set*** **(page 622)** Remove a member from a replica set.

Continue reading from *Replica Set Deployment Tutorials* (page 606) for additional tutorials of related to setting up replica set deployments.

Member Configuration Tutorials **(page 625)** Tutorials that describe the process for configuring replica set members.

- ***Adjust Priority for Replica Set Member*** **(page 625)** Change the precedence given to a replica set members in an election for primary.
- ***Prevent Secondary from Becoming Primary*** **(page 626)** Make a secondary member ineligible for election as primary.
- ***Configure a Hidden Replica Set Member*** **(page 628)** Configure a secondary member to be invisible to applications in order to support significantly different usage, such as a dedicated backups.

Continue reading from *Member Configuration Tutorials* (page 625) for more tutorials that describe replica set configuration.

Replica Set Maintenance Tutorials **(page 634)** Procedures and tasks for common operations on active replica set deployments.

- ***Change the Size of the Oplog*** **(page 634)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.
- ***Resync a Member of a Replica Set*** **(page 640)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.
- ***Force a Member to Become Primary*** **(page 638)** Force a replica set member to become primary.
- ***Change Hostnames in a Replica Set*** **(page 649)** Update the replica set configuration to reflect changes in members' hostnames.

Continue reading from *Replica Set Maintenance Tutorials* (page 634) for descriptions of additional replica set maintenance procedures.

Troubleshoot Replica Sets **(page 654)** Describes common issues and operational challenges for replica sets. For additional diagnostic information, see *FAQ: MongoDB Diagnostics* (page 799).

9.3.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.

See also:

Security Deployment Tutorials (page 348) for additional related tutorials.

Deploy a Replica Set **(page 607)** Configure a three-member replica set for production systems.

Deploy a Replica Set for Testing and Development **(page 610)** Configure a three-member replica set for either development or testing systems.

Deploy a Geographically Redundant Replica Set **(page 612)** Create a geographically redundant replica set to protect against location-centered availability limitations (e.g. network and power interruptions).

Add an Arbiter to Replica Set **(page 618)** Add an arbiter give a replica set an odd number of voting members to prevent election ties.

Convert a Standalone to a Replica Set **(page 619)** Convert an existing standalone `mongod` instance into a three-member replica set.

Add Members to a Replica Set **(page 620)** Add a new member to an existing replica set.

Remove Members from Replica Set **(page 622)** Remove a member from a replica set.

Replace a Replica Set Member **(page 624)** Update the replica set configuration when the hostname of a member's corresponding `mongod` instance has changed.

Deploy a Replica Set

On this page

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with *access control* (page 320) disabled.

To deploy a replica set with enabled *access control* (page 320), see *Deploy Replica Set and Configure Authentication and Authorization* (page 348). If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 619). For more information on replica set deployments, see the *Replication* (page 563) and *Replica Set Deployment Architectures* (page 575) documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that *elections* (page 583) will proceed smoothly. For more about designing replica sets, see *the Replication overview* (page 563).

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For production deployments, you should maintain as much separation between members as possible by hosting the `mongod` instances on separate machines. When using virtual machines for production deployments, you should place

each mongod instance on a separate host server serviced by redundant power circuits and redundant network paths.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials* (page 5).

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 655).

Considerations When Deploying a Replica Set

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of 27017. Use the bind_ip option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's mongod instances reside in the primary site.

See *Replica Set Deployment Architectures* (page 575) for more information.

Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' /etc/hosts file to reflect this configuration.

Configuration Specify the run time configuration on each system in a configuration file stored in /etc/mongod.conf or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see http://docs.mongodb.org/manual/reference/configuration-options.

Procedure

The following procedure outlines the steps to deploy a replica set when access control is disabled.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a mongod and specify the replica set name through the replSet option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *--replSet* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the `--config` option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 659):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

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

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

See also:

Deploy Replica Set and Configure Authentication and Authorization (page 348)

Deploy a Replica Set for Testing and Development

On this page

This procedure describes deploying a replica set in a development or test environment. For a production deployment, refer to the *Deploy a Replica Set* (page 607) tutorial.

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances running with *access control* (page 320) disabled.

To deploy a replica set with enabled *access control* (page 320), see *Deploy Replica Set and Configure Authentication and Authorization* (page 348). If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 619). For more information on replica set deployments, see the *Replication* (page 563) and *Replica Set Deployment Architectures* (page 575) documentation.

Overview

Three member *replica sets* provide enough redundancy to survive most network partitions and other system failures. These sets also have sufficient capacity for many distributed read operations. Replica sets should always have an odd number of members. This ensures that *elections* (page 583) will proceed smoothly. For more about designing replica sets, see *the Replication overview* (page 563).

The basic procedure is to start the `mongod` instances that will become members of the replica set, configure the replica set itself, and then add the `mongod` instances to it.

Requirements

For test and development systems, you can run your `mongod` instances on a local system, or within a virtual instance.

Before you can deploy a replica set, you must install MongoDB on each system that will be part of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials* (page 5).

Before creating your replica set, you should verify that your network configuration allows all possible connections between each member. For a successful replica set deployment, every member must be able to connect to every other member. For instructions on how to check your connection, see *Test Connections Between all Members* (page 655).

Considerations

Replica Set Naming

Important: These instructions should only be used for test or development deployments.

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

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

You will begin by starting three `mongod` instances as members of a replica set named `rs0`.

Procedure

1. Create the necessary data directories for each member by issuing a command similar to the following:

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

   This will create directories called "rs0-0", "rs0-1", and "rs0-2", which will contain the instances' database files.

2. Start your `mongod` instances in their own shell windows by issuing the following commands:

   First member:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/rs0-0 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Second member:

   ```
   mongod --port 27018 --dbpath /srv/mongodb/rs0-1 --replSet rs0 --smallfiles --oplogSize 128
   ```

   Third member:

   ```
   mongod --port 27019 --dbpath /srv/mongodb/rs0-2 --replSet rs0 --smallfiles --oplogSize 128
   ```

   This starts each instance as a member of a replica set named `rs0`, each running on a distinct port, and specifies the path to your data directory with the *`--dbpath`* setting. If you are already using the suggested ports, select different ports.

   The *`--smallfiles`* and *`--oplogSize`* settings reduce the disk space that each `mongod` instance uses. This is ideal for testing and development deployments as it prevents overloading your machine. For more information on these and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

3. Connect to one of your `mongod` instances through the `mongo` shell. You will need to indicate which instance by specifying its port number. For the sake of simplicity and clarity, you may want to choose the first one, as in the following command;

   ```
   mongo --port 27017
   ```

4. In the `mongo` shell, use `rs.initiate()` to initiate the replica set. You can create a replica set configuration object in the `mongo` shell environment, as in the following example:

   ```
   rsconf = {
              _id: "rs0",
              members: [
                         {
                          _id: 0,
                          host: "<hostname>:27017"
                         }
                       ]
            }
   ```

   replacing `<hostname>` with your system's hostname, and then pass the `rsconf` file to `rs.initiate()` as follows:

   ```
   rs.initiate( rsconf )
   ```

5. Display the current *replica configuration* (page 659) by issuing the following command:

   ```
   rs.conf()
   ```

   The replica set configuration object resembles the following

```
{
   "_id" : "rs0",
   "version" : 4,
   "members" : [
      {
         "_id" : 1,
         "host" : "localhost:27017"
      }
   ]
}
```

6. In the `mongo` shell connected to the *primary*, add the second and third `mongod` instances to the replica set using the `rs.add()` method. Replace `<hostname>` with your system's hostname in the following examples:

   ```
   rs.add("<hostname>:27018")
   rs.add("<hostname>:27019")
   ```

   When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Check the status of your replica set at any time with the `rs.status()` operation.

See also:

The documentation of the following shell functions for more information:

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

You may also consider the simple setup script[16] as an example of a basic automatically-configured replica set.

Refer to *Replica Set Read and Write Semantics* (page 588) for a detailed explanation of read and write semantics in MongoDB.

Deploy a Geographically Redundant Replica Set

On this page

Overview

This tutorial outlines the process for deploying a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

For appropriate background, see *Replication* (page 563) and *Replica Set Deployment Architectures* (page 575). For related tutorials, see *Deploy a Replica Set* (page 607) and *Add Members to a Replica Set* (page 620).

[16] https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py

Considerations

While *replica sets* provide basic protection against single-instance failure, replica sets whose members are all located in a single facility are susceptible to errors in that facility. Power outages, network interruptions, and natural disasters are all issues that can affect replica sets whose members are colocated. To protect against these classes of failures, deploy a replica set with one or more members in a geographically distinct facility or data center to provide redundancy.

Prerequisites

In general, the requirements for any geographically redundant replica set are as follows:

- Ensure that a majority of the *voting members* (page 586) are within a primary facility, "Site A". This includes *priority 0 members* (page 570) and *arbiters* (page 574). Deploy other members in secondary facilities, "Site B", "Site C", etc., to provide additional copies of the data. See *Determine the Distribution of Members* (page 576) for more information on the voting requirements for geographically redundant replica sets.
- If you deploy a replica set with an even number of members, deploy an *arbiter* (page 574) on Site A. The arbiter must be on site A to keep the majority there.

For instance, for a three-member replica set you need two instances in a Site A, and one member in a secondary facility, Site B. Site A should be the same facility or very close to your primary application infrastructure (i.e. application servers, caching layer, users, etc.)

A four-member replica set should have at least two members in Site A, with the remaining members in one or more secondary sites, as well as a single *arbiter* in Site A.

For all configurations in this tutorial, deploy each replica set member on a separate system. Although you may deploy more than one replica set member on a single system, doing so reduces the redundancy and capacity of the replica set. Such deployments are typically for testing purposes and beyond the scope of this tutorial.

This tutorial assumes you have installed MongoDB on each system that will be part of your replica set. If you have not already installed MongoDB, see the *installation tutorials* (page 5).

Procedures

General Considerations

Architecture In a production, deploy each member of the replica set to its own machine and if possible bind to the standard MongoDB port of `27017`. Use the `bind_ip` option to ensure that MongoDB listens for connections from applications on configured addresses.

For a geographically distributed replica sets, ensure that the majority of the set's `mongod` instances reside in the primary site.

See *Replica Set Deployment Architectures* (page 575) for more information.

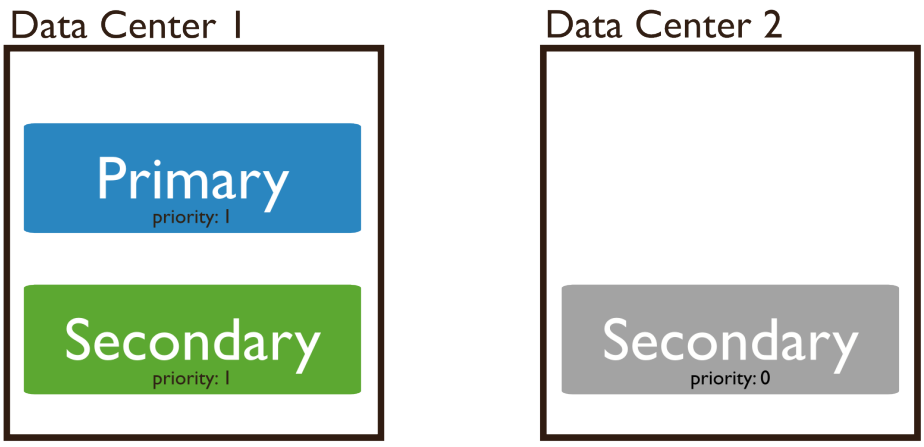
Connectivity Ensure that network traffic can pass between all members of the set and all clients in the network securely and efficiently. Consider the following:

- Establish a virtual private network. Ensure that your network topology routes all traffic between members within a single site over the local area network.
- Configure access control to prevent connections from unknown clients to the replica set.
- Configure networking and firewall rules so that incoming and outgoing packets are permitted only on the default MongoDB port and only from within your deployment.

Finally ensure that each member of a replica set is accessible by way of resolvable DNS or hostnames. You should either configure your DNS names appropriately or set up your systems' `/etc/hosts` file to reflect this configuration.

Configuration Specify the run time configuration on each system in a `configuration file` stored in `/etc/mongod.conf` or a related location. Create the directory where MongoDB stores data files before deploying MongoDB.

For more information about the run time options used above and other configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.



Deploy a Geographically Redundant Three-Member Replica Set

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 659):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

The following example adds two members:

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

When complete, you have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Configure the outside member as *priority 0 members*. Configure the member located in Site B (in this example, `mongodb2.example.net`) as a *priority 0 member* (page 570).

1. View the replica set configuration to determine the `members` (page 661) array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

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

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

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

Step 7: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Deploy a Geographically Redundant Four-Member Replica Set A geographically redundant four-member deployment has two additional considerations:

- One host (e.g. `mongodb4.example.net`) must be an *arbiter*. This host can run on a system that is also used for an application server or on the same machine as another MongoDB process.
- You must decide how to distribute your systems. There are three possible architectures for the four-member replica set:
  - Three members in Site A, one *priority 0 member* (page 570) in Site B, and an arbiter in Site A.
  - Two members in Site A, two *priority 0 members* (page 570) in Site B, and an arbiter in Site A.
  - Two members in Site A, one priority 0 member in Site B, one priority 0 member in Site C, and an arbiter in site A.

  In most cases, the first architecture is preferable because it is the least complex.

To deploy a geographically redundant four-member set:

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod` and specify the replica set name through the `replSet` option. Specify any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

The following example specifies the replica set name through the *`--replSet`* command-line option:

```
mongod --replSet "rs0"
```

You can also specify the `replica set name` in the `configuration file`. To start `mongod` with a configuration file, specify the file with the *`--config`* option:

```
mongod --config $HOME/.mongodb/config
```

In production deployments, you can configure a *control script* to manage this process. Control scripts are beyond the scope of this document.

Step 2: Connect a `mongo` shell to a replica set member. For example, to connect to a `mongod` running on localhost on the default port of `27017`, simply issue:

```
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```
rs.initiate()
```

MongoDB initiates a set that consists of the current member and that uses the default replica set configuration.

Step 4: Verify the initial replica set configuration. Use `rs.conf()` to display the *replica set configuration object* (page 659):

```
rs.conf()
```

The replica set configuration object resembles the following:

```
{
   "_id" : "rs0",
   "version" : 1,
   "members" : [
      {
         "_id" : 1,
         "host" : "mongodb0.example.net:27017"
      }
   ]
}
```

Step 5: Add the remaining members to the replica set. Use `rs.add()` in a `mongo` shell connected to the current primary. The commands should resemble the following:

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

When complete, you should have a fully functional replica set. The new replica set will elect a *primary*.

Step 6: Add the arbiter. In the same shell session, issue the following command to add the arbiter (e.g. `mongodb4.example.net`):

```
rs.addArb("mongodb4.example.net")
```

Step 7: Configure outside members as *priority 0 members*. Configure each member located outside of Site A (e.g. `mongodb3.example.net`) as a *priority 0 member* (page 570).

1. View the replica set configuration to determine the `members` (page 661) array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to `cfg` in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:

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

   Note: The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

After these commands return, you have a geographically redundant four-member replica set.

Step 8: Check the status of the replica set. Use the `rs.status()` operation:

```
rs.status()
```

Deploy a Geographically Redundant Set with More than Four Members The above procedures detail the steps necessary for deploying a geographically redundant replica set. Larger replica set deployments follow the same steps, but have additional considerations:

- Never deploy more than seven voting members.
- If you have an even number of members, use *the procedure for a four-member set* (page 616)). Ensure that a single facility, "Site A", always has a majority of the members by deploying the *arbiter* in that site. For example, if a set has six members, deploy at least three voting members in addition to the arbiter in Site A, and the remaining members in alternate sites.
- If you have an odd number of members, use *the procedure for a three-member set* (page 614). Ensure that a single facility, "Site A" always has a majority of the members of the set. For example, if a set has five members, deploy three members within Site A and two members in other facilities.
- If you have a majority of the members of the set *outside* of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.

Add an Arbiter to Replica Set

On this page

Arbiters are `mongod` instances that are part of a *replica set* but do not hold data. Arbiters participate in *elections* (page 583) in order to break ties. If a replica set has an even number of members, add an arbiter.

Arbiters have minimal resource requirements and do not require dedicated hardware. You can deploy an arbiter on an application server or a monitoring host.

Important: Do not run an arbiter on the same system as a member of the replica set.

Considerations

An arbiter does not store data, but until the arbiter's `mongod` process is added to the replica set, the arbiter will act like any other `mongod` process and start up with a set of data files and with a full-sized *journal*.

To minimize the default creation of data, set the following in the arbiter's `configuration file`:

- `journal.enabled` to `false`

  Warning: Never set `journal.enabled` to `false` on a data-bearing node.

- `smallFiles` to `true`

These settings are specific to arbiters. Do not set `journal.enabled` to `false` on a data-bearing node. Similarly, do not set `smallFiles` unless specifically indicated.

Add an Arbiter

1. Create a data directory (e.g. `dbPath`) for the arbiter. The `mongod` instance uses the directory for configuration data. The directory *will not* hold the data set. For example, create the `/data/arb` directory:

   ```
   mkdir /data/arb
   ```

2. Start the arbiter. Specify the data directory and the replica set name. The following, starts an arbiter using the `/data/arb` `dbPath` for the `rs` replica set:

   ```
   mongod --port 30000 --dbpath /data/arb --replSet rs
   ```

3. Connect to the primary and add the arbiter to the replica set. Use the `rs.addArb()` method, as in the following example:

   ```
   rs.addArb("m1.example.net:30000")
   ```

   This operation adds the arbiter running on port `30000` on the `m1.example.net` host.

Convert a Standalone to a Replica Set

On this page

This tutorial describes the process for converting a *standalone* `mongod` instance into a three-member *replica set*. Use standalone instances for testing and development, but always use replica sets in production. To install a standalone instance, see the *installation tutorials* (page 5).

To deploy a replica set without using a pre-existing `mongod` instance, see *Deploy a Replica Set* (page 607).

Procedure

1. Shut down the *standalone* `mongod` instance.
2. Restart the instance. Use the *`--replSet`* option to specify the name of the new replica set.

   For example, the following command starts a standalone instance as a member of a new replica set named `rs0`. The command uses the standalone's existing database path of `/srv/mongodb/db0`:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   If your application connects to more than one replica set, each set should have a distinct name. Some drivers group replica set connections by replica set name.

   For more information on configuration options, see `http://docs.mongodb.org/manual/reference/configuratio` and the `mongod` manual page.
3. Connect to the `mongod` instance.
4. Use `rs.initiate()` to initiate the new replica set:

   ```
   rs.initiate()
   ```

   The replica set is now operational.

   To view the replica set configuration, use `rs.conf()`. To check the status of the replica set, use `rs.status()`.

Expand the Replica Set Add additional replica set members by doing the following:

1. On two distinct systems, start two new standalone `mongod` instances. For information on starting a standalone instance, see the *installation tutorial* (page 5) specific to your environment.
2. On your connection to the original `mongod` instance (the former standalone instance), issue a command in the following form for each new instance to add to the replica set:

   ```
   rs.add("<hostname><:port>")
   ```

   Replace `<hostname>` and `<port>` with the resolvable hostname and port of the `mongod` instance to add to the set. For more information on adding a host to a replica set, see *Add Members to a Replica Set* (page 620).

Sharding Considerations If the new replica set is part of a *sharded cluster*, change the shard host information in the *config database* by doing the following:

1. Connect to one of the sharded cluster's `mongos` instances and issue a command in the following form:

   ```
   db.getSiblingDB("config").shards.save( {_id: "<name>", host: "<replica-set>/<member,><member,><.
   ```

   Replace `<name>` with the name of the shard. Replace `<replica-set>` with the name of the replica set. Replace `<member,><member,><>` with the list of the members of the replica set.
2. Restart all `mongos` instances. If possible, restart all components of the replica sets (i.e., all `mongos` and all shard `mongod` instances).

Add Members to a Replica Set

On this page

Overview

This tutorial explains how to add an additional member to an existing *replica set*. For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 575) document.

Maximum Voting Members A replica set can have a maximum of seven *voting members* (page 583). 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 586) or remove a vote from an `existing member` (page 663).

Control Scripts In production deployments you can configure a *control script* to manage member processes.

Existing Members You can use these procedures to add new members to an existing set. You can also use the same procedure to "re-add" a removed member. If the removed member's data is still relatively recent, it can recover and catch up easily.

Data Files If you have a backup or snapshot of an existing member, you can move the data files (e.g. the `dbPath` directory) to a new system and use them to quickly initiate a new member. The files must be:

- A valid copy of the data files from a member of the same replica set. See *Backup and Restore with Filesystem Snapshots* (page 256) document for more information.

  Important: Always use filesystem snapshots to create a copy of a member of the existing replica set. **Do not** use `mongodump` and `mongorestore` to seed a new replica set member.

- More recent than the oldest operation in the *primary's oplog*. The new member must be able to become current by applying operations from the primary's oplog.

Requirements

1. An active replica set.
2. A new MongoDB system capable of supporting your data set, accessible by the active replica set through the network.

Otherwise, use the MongoDB *installation tutorial* (page 5) and the *Deploy a Replica Set* (page 607) 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 597). 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 640).

  Use `rs.printReplicationInfo()` to check the current state of replica set members with regards to the oplog.

For background on replication deployment patterns, see the *Replica Set Deployment Architectures* (page 575) document.

Add a Member to an Existing Replica Set

1. Start the new `mongod` instance. Specify the data directory and the replica set name. The following example specifies the `/srv/mongodb/db0` data directory and the `rs0` replica set:

   ```
   mongod --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   Take note of the host name and port information for the new `mongod` instance.

   For more information on configuration options, see the `mongod` manual page.

Optional

You can specify the data directory and replica set in the `mongod.conf` `configuration file`, and start the `mongod` with the following command:

```
mongod --config /etc/mongod.conf
```

2. Connect to the replica set's primary.

   You can only add members while connected to the primary. If you do not know which member is the primary, log into any member of the replica set and issue the `db.isMaster()` command.

3. Use `rs.add()` to add the new member to the replica set. For example, to add a member at host `mongodb3.example.net`, issue the following command:

   ```
   rs.add("mongodb3.example.net")
   ```

   You can include the port number, depending on your setup:

   ```
   rs.add("mongodb3.example.net:27017")
   ```

4. Verify that the member is now part of the replica set. Call the `rs.conf()` method, which displays the *replica set configuration* (page 659):

   ```
   rs.conf()
   ```

   To view replica set status, issue the `rs.status()` method. For a description of the status fields, see `http://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Configure and Add a Member You can add a member to a replica set by passing to the `rs.add()` method a `members` (page 661) document. The document must be in the form of a `local.system.replset.members` (page 661) document. These documents define a replica set member in the same form as the *replica set configuration document* (page 660).

Important: Specify a value for the `_id` field of the `members` (page 661) document. MongoDB does not automatically populate the `_id` field in this case. Finally, the `members` (page 661) 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 661) of `mongodb3.example.net:27017`.
- a `priority` (page 662) value within the replica set of `0`.
- a configuration as `hidden` (page 662),

Issue the following:

```
rs.add({_id: 1, host: "mongodb3.example.net:27017", priority: 0, hidden: true})
```

Remove Members from Replica Set

On this page

To remove a member of a *replica set* use either of the following procedures.

Remove a Member Using `rs.remove()`

1. Shut down the `mongod` instance for the member you wish to remove. To shut down the instance, connect using the `mongo` shell and the `db.shutdownServer()` method.
2. Connect to the replica set's current *primary*. To determine the current primary, use `db.isMaster()` while connected to any member of the replica set.
3. Use `rs.remove()` in either of the following forms to remove the member:

   ```
   rs.remove("mongod3.example.net:27017")
   rs.remove("mongod3.example.net")
   ```

   MongoDB disconnects the shell briefly as the replica set elects a new primary. The shell then automatically reconnects. The shell displays a `DBClientCursor::init call() failed` error even though the command succeeds.

Remove a Member Using `rs.reconfig()`

To remove a member you can manually edit the *replica set configuration document* (page 659), as described here.

1. Shut down the `mongod` instance for the member you wish to remove. To shut down the instance, connect using the `mongo` shell and the `db.shutdownServer()` method.
2. Connect to the replica set's current *primary*. To determine the current primary, use `db.isMaster()` while connected to any member of the replica set.
3. Issue the `rs.conf()` method to view the current configuration document and determine the position in the `members` array of the member to remove:

   Example

   `mongod_C.example.net` is in position `2` of the following configuration file:

   ```
   {
       "_id" : "rs",
       "version" : 7,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongod_A.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongod_B.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongod_C.example.net:27017"
   ```

```
        }
    ]
}
```

4. Assign the current configuration document to the variable `cfg`:

   ```
   cfg = rs.conf()
   ```

5. Modify the `cfg` object to remove the member.

   Example

   To remove `mongod_C.example.net:27017` use the following JavaScript operation:

   ```
   cfg.members.splice(2,1)
   ```

6. Overwrite the replica set configuration document with the new configuration by issuing the following:

   ```
   rs.reconfig(cfg)
   ```

   As a result of `rs.reconfig()` the shell will disconnect while the replica set renegotiates which member is primary. The shell displays a `DBClientCursor::init call() failed` error even though the command succeeds, and will automatically reconnected.

7. To confirm the new configuration, issue `rs.conf()`.

   For the example above the output would be:

   ```
   {
       "_id" : "rs",
       "version" : 8,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongod_A.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongod_B.example.net:27017"
           }
       ]
   }
   ```

Replace a Replica Set Member

On this page

If you need to change the hostname of a replica set member without changing the configuration of that member or the set, you can use the operation outlined in this tutorial. For example if you must re-provision systems or rename hosts, you can use this pattern to minimize the scope of that change.

Operation

To change the hostname for a replica set member modify the `host` (page 661) field. The value of `_id` (page 661) field will not change when you reconfigure the set.

See *Replica Set Configuration* (page 659) and `rs.reconfig()` for more information.

Note: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 583). 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)
```

9.3.2 Member Configuration Tutorials

The following tutorials provide information in configuring replica set members to support specific operations, such as to provide dedicated backups, to support reporting, or to act as a cold standby.

Adjust Priority for Replica Set Member **(page 625)** Change the precedence given to a replica set members in an election for primary.

Prevent Secondary from Becoming Primary **(page 626)** Make a secondary member ineligible for election as primary.

Configure a Hidden Replica Set Member **(page 628)** 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 629)** 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 631)** 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 632)** Convert a secondary to an arbiter.

Adjust Priority for Replica Set Member

On this page

Overview

The priority settings of replica set members affect the outcomes of *elections* (page 583) for primary. Use this setting to ensure that some members are more likely to become primary and that others can never become primary.

The value of the member's `priority` (page 662) setting determines the member's priority in elections. The higher the number, the higher the priority.

Considerations

To modify priorities, you update the `members` (page 661) array in the replica configuration object. The array index begins with `0`. Do **not** confuse this index value with the value of the replica set member's `_id` (page 661) field in the array.

The value of `priority` (page 662) can be any floating point (i.e. decimal) number between `0` and `1000`. The default value for the `priority` (page 662) field is `1`.

To block a member from seeking election as primary, assign it a priority of `0`. *Hidden members* (page 572), *delayed members* (page 573), and *arbiters* (page **??**) all have `priority` (page 662) set to `0`.

Adjust priority during a scheduled maintenance window. Reconfiguring priority can force the current primary to step down, leading to an election. Before an election the primary closes all open *client* connections.

Procedure

Step 1: Copy the replica set configuration to a variable. In the `mongo` shell, use `rs.conf()` to retrieve the replica set configuration and assign it to a variable. For example:

```
cfg = rs.conf()
```

Step 2: Change each member's priority value. Change each member's `priority` (page 662) value, as configured in the `members` (page 661) array.

```
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
```

This sequence of operations modifies the value of `cfg` to set the priority for the first three members defined in the `members` (page 661) array.

Step 3: Assign the replica set the new configuration. Use `rs.reconfig()` to apply the new configuration.

```
rs.reconfig(cfg)
```

This operation updates the configuration of the replica set using the configuration defined by the value of `cfg`.

Prevent Secondary from Becoming Primary

On this page

Overview

In a replica set, by default all *secondary* members are eligible to become primary through the election process. You can use the `priority` (page 662) to affect the outcome of these elections by making some members more likely to become primary and other members less likely or unable to become primary.

Secondaries that cannot become primary are also unable to trigger elections. In all other respects these secondaries are identical to other secondaries.

To prevent a *secondary* member from ever becoming a *primary* in a *failover*, assign the secondary a priority of `0`, as described here. For a detailed description of secondary-only members and their purposes, see *Priority 0 Replica Set Members* (page 570).

Considerations

When updating the replica configuration object, access the replica set members in the `members` (page 661) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 661) field in each document in the `members` (page 661) array.

Note: MongoDB does not permit the current *primary* to have a priority of `0`. To prevent the current primary from again becoming a primary, you must first step down the current primary using `rs.stepDown()`.

Procedure

This tutorial uses a sample replica set with 5 members.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 583). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 618) to ensure that members can quickly obtain a majority of votes in an election for primary.

Step 1: Retrieve the current replica set configuration. The `rs.conf()` method returns a *replica set configuration document* (page 659) that contains the current configuration for a replica set.

In a `mongo` shell connected to a primary, run the `rs.conf()` method and assign the result to a variable:

```
cfg = rs.conf()
```

The returned document contains a `members` (page 661) field which contains an array of member configuration documents, one document for each member of the replica set.

Step 2: Assign priority value of 0. To prevent a secondary member from becoming a primary, update the secondary member's `priority` (page 662) to `0`.

To assign a priority value to a member of the replica set, access the member configuration document using the array index. In this tutorial, the secondary member to change corresponds to the configuration document found at position `2` of the `members` (page 661) array.

```
cfg.members[2].priority = 0
```

The configuration change does not take effect until you reconfigure the replica set.

Step 3: Reconfigure the replica set. Use `rs.reconfig()` method to reconfigure the replica set with the updated replica set configuration document.

Pass the `cfg` variable to the `rs.reconfig()` method:

```
rs.reconfig(cfg)
```

Related Documents

- `priority` (page 662)
- *Adjust Priority for Replica Set Member* (page 625)
- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 583)

Configure a Hidden Replica Set Member

On this page

Hidden members are part of a *replica set* but cannot become *primary* and are invisible to client applications. Hidden members may vote in *elections* (page 583). For a more information on hidden members and their uses, see *Hidden Replica Set Members* (page 572).

Considerations

The most common use of hidden nodes is to support *delayed members* (page 573). If you only need to prevent a member from becoming primary, configure a *priority 0 member* (page 570).

If the `chainingAllowed` (page 663) setting allows secondary members to sync from other secondaries, MongoDB by default prefers non-hidden members over hidden members when selecting a sync target. MongoDB will only choose hidden members as a last resort. If you want a secondary to sync from a hidden member, use the `replSetSyncFrom` database command to override the default sync target. See the documentation for `replSetSyncFrom` before using the command.

See also:

Manage Chained Replication (page 647)

Changed in version 2.0: For *sharded clusters* running with replica sets before 2.0, if you reconfigured a member as hidden, you *had* to restart `mongos` to prevent queries from reaching the hidden member.

Examples

Member Configuration Document To configure a secondary member as hidden, set its `priority` (page 662) value to `0` and set its `hidden` (page 662) value to `true` in its member configuration:

```
{
  "_id" : <num>
  "host" : <hostname:port>,
  "priority" : 0,
  "hidden" : true
}
```

Configuration Procedure The following example hides the secondary member currently at the index `0` in the `members` (page 661) array. To configure a *hidden member*, use the following sequence of operations in a `mongo` shell connected to the primary, specifying the member to configure by its array index in the `members` (page 661) array:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
rs.reconfig(cfg)
```

After re-configuring the set, this secondary member has a priority of `0` so that it cannot become primary and is hidden. The other members in the set will not advertise the hidden member in the `isMaster` or `db.isMaster()` output.

When updating the replica configuration object, access the replica set members in the `members` (page 661) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 661) field in each document in the `members` (page 661) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 583). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 618) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 583)
- *Read Preference* (page 591)

Configure a Delayed Replica Set Member

On this page

To configure a delayed secondary member, set its `priority` (page 662) value to `0`, its `hidden` (page 662) value to `true`, and its `slaveDelay` (page 663) value to the number of seconds to delay.

Important: The length of the secondary `slaveDelay` (page 663) must fit within the window of the oplog. If the oplog is shorter than the `slaveDelay` (page 663) window, the delayed member cannot successfully replicate operations.

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 573).

Example

The following example sets a 1-hour delay on a secondary member currently at the index `0` in the `members` (page 661) array. To set the delay, issue the following sequence of operations in a `mongo` shell connected to the primary:

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

After the replica set reconfigures, the delayed secondary member cannot become *primary* and is hidden from applications. The `slaveDelay` (page 663) 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 661) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 661) field in each document in the `members` (page 661) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 583). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 618) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `slaveDelay` (page 663)
- *Replica Set Reconfiguration*
- *Oplog Size* (page 597)
- *Change the Size of the Oplog* (page 634) tutorial
- *Replica Set Elections* (page 583)

Configure Non-Voting Replica Set Member

On this page

Non-voting members allow you to add additional members for read distribution beyond the maximum seven voting members. To configure a member as non-voting, set its `votes` (page 663) value to `0`.

Example

To disable the ability to vote in elections for the fourth, fifth, and sixth replica set members, use the following command sequence in the `mongo` shell connected to the primary. You identify each replica set member by its array index in the `members` (page 661) 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 661) array in the output of `rs.conf()`. This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

When updating the replica configuration object, access the replica set members in the `members` (page 661) array with the **array index**. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` (page 661) field in each document in the `members` (page 661) array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 583). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, try to make these changes during scheduled maintenance periods.
- To successfully reconfigure a replica set, a majority of the members must be accessible. If your replica set has an even number of members, add an *arbiter* (page 618) 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 662) to control which members are more likely to become primary.

Related Documents

- `votes` (page 663)
- *Replica Set Reconfiguration*
- *Replica Set Elections* (page 583)

Convert a Secondary to an Arbiter

On this page

If you have a *secondary* in a *replica set* that no longer needs to hold data but that needs to remain in the set to ensure that the set can *elect a primary* (page 583), 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 632).

- 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 633).

Convert Secondary to Arbiter and Reuse the Port Number

1. If your application is connecting directly to the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Shut down the secondary.
3. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method. Perform this operation while connected to the current *primary* in the `mongo` shell:

   ```
   rs.remove("<hostname><:port>")
   ```

4. Verify that the replica set no longer includes the secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

5. Move the secondary's data directory to an archive folder. For example:

   ```
   mv /data/db /data/db-old
   ```

   Optional

   You may remove the data instead.

6. Create a new, empty data directory to point to when restarting the `mongod` instance. You can reuse the previous name. For example:

   ```
   mkdir /data/db
   ```

7. Restart the `mongod` instance for the secondary, specifying the port number, the empty data directory, and the replica set. You can use the same port number you used before. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db --replSet rs
   ```

8. In the `mongo` shell convert the secondary to an arbiter using the `rs.addArb()` method:

```
rs.addArb("<hostname><:port>")
```

9. Verify the arbiter belongs to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

Convert Secondary to Arbiter Running on a New Port Number

1. If your application is connecting directly to the secondary or has a connection string referencing the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Create a new, empty data directory to be used with the new port number. For example:

   ```
   mkdir /data/db-temp
   ```

3. Start a new `mongod` instance on the new port number, specifying the new data directory and the existing replica set. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db-temp --replSet rs
   ```

4. In the `mongo` shell connected to the current primary, convert the new `mongod` instance to an arbiter using the `rs.addArb()` method:

   ```
   rs.addArb("<hostname><:port>")
   ```

5. Verify the arbiter has been added to the replica set by calling the `rs.conf()` method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

6. Shut down the secondary.
7. Remove the *secondary* from the *replica set* by calling the `rs.remove()` method in the `mongo` shell:

   ```
   rs.remove("<hostname><:port>")
   ```

8. Verify that the replica set no longer includes the old secondary by calling the `rs.conf()` method in the `mongo` shell:

   ```
   rs.conf()
   ```

9. Move the secondary's data directory to an archive folder. For example:

   ```
   mv /data/db /data/db-old
   ```

   Optional

   You may remove the data instead.

9.3.3 Replica Set Maintenance Tutorials

The following tutorials provide information in maintaining existing replica sets.

Change the Size of the Oplog **(page 634)** Increase the size of the *oplog* which logs operations. In most cases, the default oplog size is sufficient.

Perform Maintenance on Replica Set Members **(page 636)** Perform maintenance on a member of a replica set while minimizing downtime.

Force a Member to Become Primary **(page 638)** Force a replica set member to become primary.

Resync a Member of a Replica Set **(page 640)** Sync the data on a member. Either perform initial sync on a new member or resync the data on an existing member that has fallen too far behind to catch up by way of normal replication.

Configure Replica Set Tag Sets **(page 641)** 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 645)** Reconfigure a replica set when a majority of replica set members are down or unreachable.

Manage Chained Replication **(page 647)** 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 649)** Update the replica set configuration to reflect changes in members' hostnames.

Configure a Secondary's Sync Target **(page 652)** Specify the member that a secondary member synchronizes from.

Change the Size of the Oplog

On this page

The *oplog* exists internally as a *capped collection*, so you cannot modify its size in the course of normal operations. In most cases the *default oplog size* (page 597) is an acceptable size; however, in some situations you may need a larger or smaller oplog. For example, you might need to change the oplog size if your applications perform large numbers of multi-updates or deletes in short periods of time.

This tutorial describes how to resize the oplog. For a detailed explanation of oplog sizing, see *Oplog Size* (page 597). For details how oplog size affects *delayed members* and affects *replication lag*, see *Delayed Replica Set Members* (page 573).

Overview

To change the size of the oplog, you must perform maintenance on each member of the replica set in turn. The procedure requires: stopping the `mongod` instance and starting as a standalone instance, modifying the oplog size, and restarting the member.

Important: Always start rolling replica set maintenance with the secondaries, and finish with the maintenance on primary member.

Procedure

- Restart the member in standalone mode.

  Tip

  Always use `rs.stepDown()` to force the primary to become a secondary, before stopping the server. This facilitates a more efficient election process.

- Recreate the oplog with the new size and with an old oplog entry as a seed.
- Restart the `mongod` instance as a member of the replica set.

Restart a Secondary in Standalone Mode on a Different Port Shut down the `mongod` instance for one of the non-primary members of your replica set. For example, to shut down, use the `db.shutdownServer()` method:

```
db.shutdownServer()
```

Restart this `mongod` as a standalone instance running on a different port and *without* the *`--replSet`* parameter. Use a command similar to the following:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Create a Backup of the Oplog (Optional) Optionally, backup the existing oplog on the standalone instance, as in the following example:

```
mongodump --db local --collection 'oplog.rs' --port 37017
```

Recreate the Oplog with a New Size and a Seed Entry Save the last entry from the oplog. For example, connect to the instance using the `mongo` shell, and enter the following command to switch to the `local` database:

```
use local
```

In `mongo` shell scripts you can use the following operation to set the `db` object:

```
db = db.getSiblingDB('local')
```

Ensure that the `temp` temporary collection is empty by dropping the collection:

```
db.temp.drop()
```

Use the `db.collection.save()` method and a sort on reverse *natural order* to find the last entry and save it to a temporary collection:

```
db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural : -1} ).limit(1).next() )
```

To see this oplog entry, use the following operation:

```
db.temp.find()
```

Remove the Existing Oplog Collection Drop the old `oplog.rs` collection in the `local` database. Use the following command:

```
db = db.getSiblingDB('local')
db.oplog.rs.drop()
```

This returns `true` in the shell.

Create a New Oplog Use the `create` command to create a new oplog of a different size. Specify the `size` argument in bytes. A value of `2 * 1024 * 1024 * 1024` will create a new oplog that's 2 gigabytes:

```
db.runCommand( { create: "oplog.rs", capped: true, size: (2 * 1024 * 1024 * 1024) } )
```

Upon success, this command returns the following status:

```
{ "ok" : 1 }
```

Insert the Last Entry of the Old Oplog into the New Oplog Insert the previously saved last entry from the old oplog into the new oplog. For example:

```
db.oplog.rs.save( db.temp.findOne() )
```

To confirm the entry is in the new oplog, use the following operation:

```
db.oplog.rs.find()
```

Restart the Member Restart the `mongod` as a member of the replica set on its usual port. For example:

```
db.shutdownServer()
mongod --replSet rs0 --dbpath /srv/mongodb
```

The replica set member will recover and "catch up" before it is eligible for election to primary.

Repeat Process for all Members that may become Primary Repeat this procedure for all members you want to change the size of the oplog. Repeat the procedure for the primary as part of the following step.

Change the Size of the Oplog on the Primary To finish the rolling maintenance operation, step down the primary with the `rs.stepDown()` method and repeat the oplog resizing procedure above.

Perform Maintenance on Replica Set Members

On this page

Overview

Replica sets allow a MongoDB deployment to remain available during the majority of a maintenance window.

This document outlines the basic procedure for performing maintenance on each of the members of a replica set. Furthermore, this particular sequence strives to minimize the amount of time that the *primary* is unavailable and controlling the impact on the entire deployment.

Use these steps as the basis for common replica set operations, particularly for procedures such as *upgrading to the latest version of MongoDB* (page 247) and *changing the size of the oplog* (page 634).

Procedure

For each member of a replica set, starting with a secondary member, perform the following sequence of events, ending with the primary:

- Restart the `mongod` instance as a standalone.
- Perform the task on the standalone instance.
- Restart the `mongod` instance as a member of the replica set.

Step 1: Stop a secondary. In the `mongo` shell, shut down the `mongod` instance:

```
db.shutdownServer()
```

Step 2: Restart the secondary as a standalone on a different port. At the operating system shell prompt, restart `mongod` as a standalone instance running on a different port and *without* the `--replSet` parameter:

```
mongod --port 37017 --dbpath /srv/mongodb
```

Always start `mongod` with the same user, even when restarting a replica set member as a standalone instance.

Step 3: Perform maintenance operations on the secondary. While the member is a standalone, use the `mongo` shell to perform maintenance:

```
mongo --port 37017
```

Step 4: Restart `mongod` as a member of the replica set. After performing all maintenance tasks, use the following procedure to restart the `mongod` as a member of the replica set on its usual port.

From the `mongo` shell, shut down the standalone server after completing the maintenance:

```
db.shutdownServer()
```

Restart the `mongod` instance as a member of the replica set using its normal command-line arguments or configuration file.

The secondary takes time to *catch up to the primary* (page 598). From the `mongo` shell, use the following command to verify that the member has caught up from the `RECOVERING` (page 668) state to the `SECONDARY` (page 667) state.

```
rs.status()
```

Step 5: Perform maintenance on the primary last. To perform maintenance on the primary after completing maintenance tasks on all secondaries, use `rs.stepDown()` in the `mongo` shell to step down the primary and allow one of the secondaries to be elected the new primary. Specify a 300 second waiting period to prevent the member from being elected primary again for five minutes:

```
rs.stepDown(300)
```

After the primary steps down, the replica set will elect a new primary. See *Replica Set Elections* (page 583) for more information about replica set elections.

Force a Member to Become Primary

On this page

Overview

You can force a *replica set* member to become *primary* by giving it a higher `priority` (page 662) value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `priority` (page 662) value to `0`, which means the member can never seek *election* (page 583) as primary. For more information, see *Priority 0 Replica Set Members* (page 570).

For more information on priorities, see `priority` (page 662).

Consideration

A majority of the configured members of a replica set *must* be available for a set to reconfigure a set or elect a primary. See *Replica Set Elections* (page 583) for more information.

Procedures

Force a Member to be Primary by Setting its Priority High This procedure assumes your current *primary* is `m1.example.net` and that you'd like to instead make `m3.example.net` primary. The procedure also assumes you have a three-member *replica set* with the configuration below. For more information on configurations, see *Replica Set Configuration Use*.

This procedure assumes this configuration:

```
{
    "_id" : "rs",
    "version" : 7,
    "members" : [
        {
            "_id" : 0,
            "host" : "m1.example.net:27017"
        },
        {
            "_id" : 1,
            "host" : "m2.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "m3.example.net:27017"
        }
    ]
}
```

1. In a mongo shell connected to the primary, use the following sequence of operations to make m3.example.net the primary:

   ```
   cfg = rs.conf()
   cfg.members[0].priority = 0.5
   cfg.members[1].priority = 0.5
   cfg.members[2].priority = 1
   rs.reconfig(cfg)
   ```

   The last statement calls rs.reconfig() with the modified configuration document to configure m3.example.net to have a higher local.system.replset.members[n].priority (page 662) value than the other mongod instances.

   The following sequence of events occur:

   - m3.example.net and m2.example.net sync with m1.example.net (typically within 10 seconds).
   - m1.example.net sees that it no longer has highest priority and, in most cases, steps down. m1.example.net *does not* step down if m3.example.net's sync is far behind. In that case, m1.example.net waits until m3.example.net is within 10 seconds of its optime and then steps down. This minimizes the amount of time with no primary following failover.
   - The step down forces on election in which m3.example.net becomes primary based on its priority (page 662) setting.

2. Optionally, if m3.example.net is more than 10 seconds behind m1.example.net's optime, and if you don't need to have a primary designated within 10 seconds, you can force m1.example.net to step down by running:

   ```
   db.adminCommand({replSetStepDown: 86400, force: 1})
   ```

   This prevents m1.example.net from being primary for 86,400 seconds (24 hours), even if there is no other member that can become primary. When m3.example.net catches up with m1.example.net it will become primary.

   If you later want to make m1.example.net primary again while it waits for m3.example.net to catch up, issue the following command to make m1.example.net seek election again:

   ```
   rs.freeze()
   ```

   The rs.freeze() provides a wrapper around the replSetFreeze database command.

Force a Member to be Primary Using Database Commands Changed in version 1.8.

Consider a *replica set* with the following members:

- mdb0.example.net - the current *primary*.
- mdb1.example.net - a *secondary*.
- mdb2.example.net - a secondary .

To force a member to become primary use the following procedure:

1. In a mongo shell, run rs.status() to ensure your replica set is running as expected.
2. In a mongo shell connected to the mongod instance running on mdb2.example.net, freeze mdb2.example.net so that it does not attempt to become primary for 120 seconds.

   ```
   rs.freeze(120)
   ```

3. In a `mongo` shell connected the `mongod` running on `mdb0.example.net`, step down this instance that the `mongod` is not eligible to become primary for 120 seconds:

   ```
   rs.stepDown(120)
   ```

   `mdb1.example.net` becomes primary.

   Note: During the transition, there is a short window where the set does not have a primary.

For more information, consider the `rs.freeze()` and `rs.stepDown()` methods that wrap the `replSetFreeze` and `replSetStepDown` commands.

Resync a Member of a Replica Set

On this page

A *replica set* member becomes "stale" when its replication process falls so far behind that the *primary* overwrites oplog entries the member has not yet replicated. The member cannot catch up and becomes "stale." When this occurs, you must completely resynchronize the member by removing its data and performing an *initial sync* (page 598).

This tutorial addressed both resyncing a stale member and to creating a new member using seed data from another member. When syncing a member, choose a time when the system has the bandwidth to move a large amount of data. Schedule the synchronization during a time of low usage or during a maintenance window.

MongoDB provides two options for performing an initial sync:

- Restart the `mongod` with an empty data directory and let MongoDB's normal initial syncing feature restore the data. This is the more simple option but may take longer to replace the data.

  See *Procedures* (page 640).

- Restart the machine with a copy of a recent data directory from another member in the replica set. This procedure can replace the data more quickly but requires more manual steps.

  See *Sync by Copying Data Files from Another Member* (page 641).

Procedures

Automatically Sync a Member

Warning: During initial sync, `mongod` will remove the content of the `dbPath`.

This procedure relies on MongoDB's regular process for *initial sync* (page 598). This will store the current data on the member. For an overview of MongoDB initial sync process, see the *Replication Processes* (page 596) section.

If the instance has no data, you can simply follow the *Add Members to a Replica Set* (page 620) or *Replace a Replica Set Member* (page 624) procedure to add a new member to a replica set.

You can also force a `mongod` that is already a member of the set to to perform an initial sync by restarting the instance without the content of the `dbPath` as follows:

1. Stop the member's `mongod` instance. To ensure a clean shutdown, use the `db.shutdownServer()` method from the `mongo` shell or on Linux systems, the *`mongod --shutdown`* option.
2. Delete all data and sub-directories from the member's data directory. By removing the data `dbPath`, MongoDB will perform a complete resync. Consider making a backup first.

At this point, the `mongod` will perform an initial sync. The length of the initial sync process depends on the size of the database and network connection between members of the replica set.

Initial sync operations can impact the other members of the set and create additional traffic to the primary and can only occur if another member of the set is accessible and up to date.

Sync by Copying Data Files from Another Member This approach "seeds" a new or stale member using the data files from an existing member of the replica set. The data files **must** be sufficiently recent to allow the new member to catch up with the *oplog*. Otherwise the member would need to perform an initial sync.

Copy the Data Files You can capture the data files as either a snapshot or a direct copy. However, in most cases you cannot copy data files from a running `mongod` instance to another because the data files will change during the file copy operation.

Important: If copying data files, you must copy the content of the `local` database.

You *cannot* use a `mongodump` backup for the data files, **only a snapshot backup**. For approaches to capturing a consistent snapshot of a running `mongod` instance, see the *MongoDB Backup Methods* (page 192) documentation.

Sync the Member After you have copied the data files from the "seed" source, start the `mongod` instance and allow it to apply all operations from the oplog until it reflects the current state of the replica set.

Configure Replica Set Tag Sets

On this page

Tag sets let you customize *write concern* and *read preferences* for a *replica set*. MongoDB stores tag sets in the replica set configuration object, which is the document returned by `rs.conf()`, in the `members[n].tags` (page 662) embedded document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see *Replica Set Write Concern* (page 83) and *Tag Sets* (page 594).

Differences Between Read Preferences and Write Concerns

Custom read preferences and write concerns evaluate tags sets in different ways:

- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns do not use the value of a tag to select a member except to consider whether or not the value is unique.

For example, a tag set for a read operation may resemble the following document:

```
{ "disk": "ssd", "use": "reporting" }
```

To fulfill such a read operation, a member would need to have both of these tags. Any of the following tag sets would satisfy this requirement:

```
{ "disk": "ssd", "use": "reporting" }
{ "disk": "ssd", "use": "reporting", "rack": "a" }
{ "disk": "ssd", "use": "reporting", "rack": "d" }
{ "disk": "ssd", "use": "reporting", "mem": "r"}
```

The following tag sets would *not* be able to fulfill this query:

```
{ "disk": "ssd" }
{ "use": "reporting" }
{ "disk": "ssd", "use": "production" }
{ "disk": "ssd", "use": "production", "rack": "k" }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Add Tag Sets to a Replica Set

Given the following replica set configuration:

```
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [
            {
                    "_id" : 0,
                    "host" : "mongodb0.example.net:27017"
            },
            {
                    "_id" : 1,
                    "host" : "mongodb1.example.net:27017"
            },
            {
                    "_id" : 2,
                    "host" : "mongodb2.example.net:27017"
            }
     ]
}
```

You could add tag sets to the members of this replica set with the following command sequence in the `mongo` shell:

```
conf = rs.conf()
conf.members[0].tags = { "dc": "east", "use": "production"  }
conf.members[1].tags = { "dc": "east", "use": "reporting"  }
conf.members[2].tags = { "use": "production"  }
rs.reconfig(conf)
```

After this operation the output of `rs.conf()` would resemble the following:

```
{
    "_id" : "rs0",
    "version" : 2,
    "members" : [
            {
                    "_id" : 0,
                    "host" : "mongodb0.example.net:27017",
                    "tags" : {
                            "dc": "east",
```

```
                        "use": "production"
                }
        },
        {
                "_id" : 1,
                "host" : "mongodb1.example.net:27017",
                "tags" : {
                        "dc": "east",
                        "use": "reporting"
                }
        },
        {
                "_id" : 2,
                "host" : "mongodb2.example.net:27017",
                "tags" : {
                        "use": "production"
                }
        }
    ]
}
```

Important: In tag sets, all tag values must be strings.

Custom Multi-Datacenter Write Concerns

Given a five member replica set with members in two data centers:

1. a facility `VA` tagged `dc_va`
2. a facility `GTO` tagged `dc_gto`

Create a custom write concern to require confirmation from two data centers using replica set tags, using the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration JavaScript object `conf`:

   ```
   conf = rs.conf()
   ```

2. Add tags to the replica set members reflecting their locations:

   ```
   conf.members[0].tags = { "dc_va": "rack1"}
   conf.members[1].tags = { "dc_va": "rack2"}
   conf.members[2].tags = { "dc_gto": "rack1"}
   conf.members[3].tags = { "dc_gto": "rack2"}
   conf.members[4].tags = { "dc_va": "rack1"}
   rs.reconfig(conf)
   ```

3. Create a custom `getLastErrorModes` (page 664) setting to ensure that the write operation will propagate to at least one member of each facility:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc_va": 1, "dc_gto": 1 } } }
   ```

4. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

To ensure that a write operation propagates to at least one member of the set in both data centers, use the `MultipleDC` write concern mode as follows:

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Alternatively, if you want to ensure that each write operation propagates to at least 2 racks in each facility, reconfigure the replica set as follows in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 664) value to require two different values of both `dc_va` and `dc_gto`:

   ```
   conf.settings = { getLastErrorModes: { MultipleDC : { "dc_va": 2, "dc_gto": 2}}
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now, the following write operation will only return after the write operation propagates to at least two different racks in the each facility:

Changed in version 2.6: A new protocol for *write operations* (page 832) integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Configure Tag Sets for Functional Segregation of Read and Write Operations

Given a replica set with tag sets that reflect:

- data center facility,
- physical rack location of instance, and
- storage system (i.e. disk) type.

Where each member of the set has a tag set that resembles one of the following: [17]

```
{"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 664) configuration. Consider the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` (page 664) value to configure two write concern modes:

[17] Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.

```
conf.settings = {
                  "getLastErrorModes" : {
                          "ssd" : {
                                      "ssd" : 1
                          },
                          "MultipleDC" : {
                                  "dc_va" : 1,
                                 "dc_gto" : 1
                          }
                  }
                }
```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```
   rs.reconfig(conf)
   ```

Now you can specify the `MultipleDC` write concern mode, as in the following, to ensure that a write operation propagates to each data center.

Changed in version 2.6: A new protocol for *write operations* (page 832) integrates write concerns with the write operations. Previous versions used the `getLastError` command to specify the write concerns.

```
db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )
```

Additionally, you can specify the `ssd` write concern mode to ensure that a write operation propagates to at least one instance with an SSD.

Reconfigure a Replica Set with Unavailable Members

On this page

To reconfigure a *replica set* when a **majority** of members are available, use the `rs.reconfig()` operation on the current *primary*, following the example in the *Replica Set Reconfiguration Procedure*.

This document provides the following options for re-configuring a replica set when *only* a **minority** of members are accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 645)
- *Reconfigure by Replacing the Replica Set* (page 646)

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 583) for more information on this situation.

Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of *replica set* members are down or unreachable. You connect to any surviving member and use the `force` option to the `rs.reconfig()` method.

The `force` option forces a new configuration onto the member. Use this procedure only to recover from catastrophic interruptions. Do not use `force` every time you reconfigure. Also, do not use the `force` option in any automatic scripts and do not use `force` when there is still a *primary*.

To force reconfiguration:

1. Back up a surviving member.
2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

   ```
   cfg = rs.conf()

   printjson(cfg)
   ```

3. On the same member, remove the down and unreachable members of the replica set from the `members` (page 661) array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

   ```
   cfg.members = [cfg.members[0] , cfg.members[4] , cfg.members[7]]
   ```

4. On the same member, reconfigure the set by using the `rs.reconfig()` command with the `force` option set to `true`:

   ```
   rs.reconfig(cfg, {force : true})
   ```

   This operation forces the secondary to use the new configuration. The configuration is then propagated to all the surviving members listed in the `members` array. The replica set then elects a new primary.

   Note: When you use `force : true`, the version number in the replica set configuration increases significantly, by tens or hundreds of thousands. This is normal and designed to prevent set version collisions if you accidentally force re-configurations on both sides of a network partition and then the network partitioning ends.

   ---

5. If the failure or partition was only temporary, shut down or decommission the removed members as soon as possible.

Reconfigure by Replacing the Replica Set

Use the following procedure **only** for versions of MongoDB prior to version 2.0. If you're running MongoDB 2.0 or later, use the above procedure, *Reconfigure by Forcing the Reconfiguration* (page 645).

These procedures are for situations where a *majority* of the *replica set* members are down or unreachable. If a majority is *running*, then skip these procedures and instead use the `rs.reconfig()` command according to the examples in *replica-set-reconfiguration-usage*.

If you run a pre-2.0 version and a majority of your replica set is down, you have the two options described here. Both involve replacing the replica set.

Reconfigure by Turning Off Replication This option replaces the *replica set* with a *standalone* server.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Create a backup of the data directory (i.e. `dbPath`) of the surviving members of the set.

   Optional

If you have a backup of the database you may instead remove this data.

3. Restart one of the `mongod` instances *without* the `--replSet` parameter.

   The data is now accessible and provided by a single server that is not a replica set member. Clients can use this server for both reads and writes.

When possible, re-deploy a replica set to provide redundancy and to protect your deployment from operational interruption.

Reconfigure by "Breaking the Mirror" This option selects a surviving *replica set* member to be the new *primary* and to "seed" a new replica set. In the following procedure, the new primary is `db0.example.net`. MongoDB copies the data from `db0.example.net` to all the other members.

1. Stop the surviving `mongod` instances. To ensure a clean shutdown, use an existing *control script* or use the `db.shutdownServer()` method.

   For example, to use the `db.shutdownServer()` method, connect to the server using the `mongo` shell and issue the following sequence of commands:

   ```
   use admin
   db.shutdownServer()
   ```

2. Move the data directories (i.e. `dbPath`) for all the members except `db0.example.net`, so that all the members except `db0.example.net` have empty data directories. For example:

   ```
   mv /data/db /data/db-old
   ```

3. Move the data files for `local` database (i.e. `local.*`) so that `db0.example.net` has no local database. For example

   ```
   mkdir /data/local-old
   mv /data/db/local* /data/local-old/
   ```

4. Start each member of the replica set normally.
5. Connect to `db0.example.net` in a `mongo` shell and run `rs.initiate()` to initiate the replica set.
6. Add the other set members using `rs.add()`. For example, to add a member running on `db1.example.net` at port `27017`, issue the following command:

   ```
   rs.add("db1.example.net:27017")
   ```

   MongoDB performs an initial sync on the added members by copying all data from `db0.example.net` to the added members.

See also:

Resync a Member of a Replica Set (page 640)

Manage Chained Replication

On this page

Starting in version 2.0, MongoDB supports chained replication. A chained replication occurs when a *secondary* member replicates from another secondary member instead of from the *primary*. This might be the case, for example, if a secondary selects its replication target based on ping time and if the closest member is another secondary.

Chained replication can reduce load on the primary. But chained replication can also result in increased replication lag, depending on the topology of the network.

New in version 2.2.2.

You can use the `chainingAllowed` (page 663) setting in *Replica Set Configuration* (page 659) to disable chained replication for situations where chained replication is causing lag.

MongoDB enables chained replication by default. This procedure describes how to disable it and how to re-enable it.

Note: If chained replication is disabled, you still can use `replSetSyncFrom` to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Disable Chained Replication

To disable chained replication, set the `chainingAllowed` (page 663) field in *Replica Set Configuration* (page 659) to `false`.

You can use the following sequence of commands to set `chainingAllowed` (page 663) to `false`:

1. Copy the configuration settings into the `cfg` object:

   ```
   cfg = rs.config()
   ```

2. Take note of whether the current configuration settings contain the `settings` embedded document. If they do, skip this step.

   Warning: To avoid data loss, skip this step if the configuration settings contain the `settings` embedded document.

   If the current configuration settings **do not** contain the `settings` embedded document, create the embedded document by issuing the following command:

   ```
   cfg.settings = { }
   ```

3. Issue the following sequence of commands to set `chainingAllowed` (page 663) to `false`:

   ```
   cfg.settings.chainingAllowed = false
   rs.reconfig(cfg)
   ```

Re-enable Chained Replication

To re-enable chained replication, set `chainingAllowed` (page 663) to `true`. You can use the following sequence of commands:

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```

Change Hostnames in a Replica Set

On this page

For most *replica sets*, the hostnames in the `host` (page 661) 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 661) 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 661) field. Use either of the following approaches:

- *Change hostnames without disrupting availability* (page 650). 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 651). This approach has a shorter maintenance window, but the replica set will be unavailable during the operation.

See also:

Replica Set Reconfiguration Process, *Deploy a Replica Set* (page 607), and *Add Members to a Replica Set* (page 620).

Assumptions

Given a *replica set* with three members:

- `database0.example.com:27017` (the *primary*)
- `database1.example.com:27017`
- `database2.example.com:27017`

And with the following `rs.conf()` output:

```
{
    "_id" : "rs",
    "version" : 3,
    "members" : [
        {
            "_id" : 0,
            "host" : "database0.example.com:27017"
        },
        {
```

```
            "_id" : 1,
            "host" : "database1.example.com:27017"
        },
        {
            "_id" : 2,
            "host" : "database2.example.com:27017"
        }
    ]
}
```

The following procedures change the members' hostnames as follows:

- `mongodb0.example.net:27017` (the primary)
- `mongodb1.example.net:27017`
- `mongodb2.example.net:27017`

Use the most appropriate procedure for your deployment.

Change Hostnames while Maintaining Replica Set Availability

This procedure uses the above *assumptions* (page 649).

1. For each *secondary* in the replica set, perform the following sequence of operations:
   (a) Stop the secondary.
   (b) Restart the secondary at the new location.
   (c) Open a `mongo` shell connected to the replica set's primary. In our example, the primary runs on port `27017` so you would issue the following command:

   ```
   mongo --port 27017
   ```

   (d) Use `rs.reconfig()` to update the *replica set configuration document* (page 659) with the new hostname.

   For example, the following sequence of commands updates the hostname for the secondary at the array index `1` of the `members` array (i.e. `members[1]`) in the replica set configuration document:

   ```
   cfg = rs.conf()
   cfg.members[1].host = "mongodb1.example.net:27017"
   rs.reconfig(cfg)
   ```

   For more information on updating the configuration document, see *replica-set-reconfiguration-usage*.
   (e) Make sure your client applications are able to access the set at the new location and that the secondary has a chance to catch up with the other members of the set.

   Repeat the above steps for each non-primary member of the set.
2. Open a `mongo` shell connected to the primary and step down the primary using the `rs.stepDown()` method:

   ```
   rs.stepDown()
   ```

   The replica set elects another member to the become primary.
3. When the step down succeeds, shut down the old primary.
4. Start the `mongod` instance that will become the new primary in the new location.

5. Connect to the current primary, which was just elected, and update the *replica set configuration document* (page 659) with the hostname of the node that is to become the new primary.

   For example, if the old primary was at position `0` and the new primary's hostname is `mongodb0.example.net:27017`, you would run:

   ```
   cfg = rs.conf()
   cfg.members[0].host = "mongodb0.example.net:27017"
   rs.reconfig(cfg)
   ```

6. Open a `mongo` shell connected to the new primary.

7. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
           }
       ]
   }
   ```

Change All Hostnames at the Same Time

This procedure uses the above *assumptions* (page 649).

1. Stop all members in the *replica set*.

2. Restart each member *on a different port* and *without* using the `--replSet` run-time option. Changing the port number during maintenance prevents clients from connecting to this host while you perform maintenance. Use the member's usual `--dbpath`, which in this example is `/data/db1`. Use a command that resembles the following:

   ```
   mongod --dbpath /data/db1/ --port 37017
   ```

3. For each member of the replica set, perform the following sequence of operations:

   (a) Open a `mongo` shell connected to the `mongod` running on the new, temporary port. For example, for a member running on a temporary port of `37017`, you would issue this command:

   ```
   mongo --port 37017
   ```

   (b) Edit the replica set configuration manually. The replica set configuration is the only document in the `system.replset` collection in the `local` database. Edit the replica set configuration with the new hostnames and correct ports for all the members of the replica set. Consider the following sequence of commands to change the hostnames in a three-member set:

```
use local

cfg = db.system.replset.findOne( { "_id": "rs" } )

cfg.members[0].host = "mongodb0.example.net:27017"

cfg.members[1].host = "mongodb1.example.net:27017"

cfg.members[2].host = "mongodb2.example.net:27017"

db.system.replset.update( { "_id": "rs" } , cfg )
```

(c) Stop the `mongod` process on the member.

4. After re-configuring all members of the set, start each `mongod` instance in the normal way: use the usual port number and use the *`--replSet`* option. For example:

   ```
   mongod --dbpath /data/db1/ --port 27017 --replSet rs
   ```

5. Connect to one of the `mongod` instances using the `mongo` shell. For example:

   ```
   mongo --port 27017
   ```

6. To confirm the new configuration, call `rs.conf()` in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
               "_id" : 1,
               "host" : "mongodb1.example.net:27017"
           },
           {
               "_id" : 2,
               "host" : "mongodb2.example.net:27017"
           }
       ]
   }
   ```

Configure a Secondary's Sync Target

On this page

Overview

Secondaries capture data from the primary member to maintain an up to date copy of the sets' data. However, by default secondaries may automatically change their sync targets to secondary members based on changes in the ping time between members and the state of other members' replication. See *Replica Set Data Synchronization* (page 598) and *Manage Chained Replication* (page 647) for more information.

For some deployments, implementing a custom replication sync topology may be more effective than the default sync target selection logic. MongoDB provides the ability to specify a host to use as a sync target.

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target to temporarily pull *oplog* entries. The following provide access to this functionality:

- `replSetSyncFrom` command, or
- `rs.syncFrom()` helper in the `mongo` shell

Considerations

Sync Logic Only modify the default sync logic as needed, and always exercise caution. `rs.syncFrom()` will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run `rs.syncFrom()` operation *before* initial sync.

If you run `rs.syncFrom()` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Persistence `replSetSyncFrom` and `rs.syncFrom()` provide a temporary override of default behavior. `mongod` will revert to the default sync behavior in the following situations:

- The `mongod` instance restarts.
- The connection between the `mongod` and the sync target closes.

Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set; the `mongod` will revert to the default sync target.

Target The member to sync from must be a valid source for data in the set. To sync from a member, the member must:

- Have data. It cannot be an arbiter, in startup or recovering mode, and must be able to answer data queries.
- Be accessible.
- Be a member of the same set in the replica set configuration.
- Build indexes with the `buildIndexes` (page 661) setting.
- A different member of the set, to prevent syncing from itself.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Procedure

To use the `replSetSyncFrom` command in the `mongo` shell:

```
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } );
```

To use the `rs.syncFrom()` helper in the `mongo` shell:

```
rs.syncFrom("hostname<:port>");
```

9.3.4 Troubleshoot Replica Sets

On this page

This section describes common strategies for troubleshooting *replica set* deployments.

Check Replica Set Status

To display the current state of the replica set and current state of each member, run the `rs.status()` method in a `mongo` shell connected to the replica set's *primary*. For descriptions of the information displayed by `rs.status()`, see `http://docs.mongodb.org/manual/reference/command/replSetGetStatus`.

Note: The `rs.status()` method is a wrapper that runs the `replSetGetStatus` database command.

Check the Replication Lag

Replication lag is a delay between an operation on the *primary* and the application of that operation from the *oplog* to the *secondary*. Replication lag can be a significant issue and can seriously affect MongoDB *replica set* deployments. Excessive replication lag makes "lagged" members ineligible to quickly become primary and increases the possibility that distributed read operations will be inconsistent.

To check the current length of replication lag:

- In a `mongo` shell connected to the primary, call the `rs.printSlaveReplicationInfo()` method.

  Returns the `syncedTo` value for each member, which shows the time when the last oplog entry was written to the secondary, as shown in the following example:

  ```
  source: m1.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  source: m2.example.net:27017
      syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
      0 secs (0 hrs) behind the primary
  ```

A *delayed member* (page 573) may show as `0` seconds behind the primary when the inactivity period on the primary is greater than the `slaveDelay` (page 663) value.

Note: The `rs.status()` method is a wrapper around the `replSetGetStatus` database command.

- Monitor the rate of replication by watching the oplog time in the "replica" graph in the MongoDB Cloud Manager[18]. For more information, see the MongoDB Cloud Manager documentation[19].

Possible causes of replication lag include:

- **Network Latency**

  Check the network routes between the members of your set to ensure that there is no packet loss or network routing issue.

  Use tools including `ping` to test latency between set members and `traceroute` to expose the routing of packets network endpoints.

- **Disk Throughput**

  If the file system and disk device on the secondary is unable to flush data to disk as quickly as the primary, then the secondary will have difficulty keeping state. Disk-related issues are incredibly prevalent on multi-tenant systems, including virtualized instances, and can be transient if the system accesses disk devices over an IP network (as is the case with Amazon's EBS system.)

  Use system-level tools to assess disk status, including `iostat` or `vmstat`.

- **Concurrency**

  In some cases, long-running operations on the primary can block replication on secondaries. For best results, configure *write concern* (page 82) to require confirmation of replication to secondaries, as described in *replica set write concern* (page 83). 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 82), 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 82) 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 83)
  - *Replica Set Write Concern* (page 88)
  - *Oplog Size* (page 597)

Test Connections Between all Members

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both "directions." Networking topologies and firewall configurations can prevent normal and required connectivity, which can block replication.

[18] https://cloud.mongodb.com/?jmp=docs
[19] https://docs.cloud.mongodb.com/

Consider the following example of a bidirectional test of networking:

Example

Given a replica set with three members running on three separate hosts:

- `m1.example.net`
- `m2.example.net`
- `m3.example.net`

1. Test the connection from `m1.example.net` to the other hosts with the following operation set `m1.example.net`:

   ```
   mongo --host m2.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

2. Test the connection from `m2.example.net` to the other two hosts with the following operation set from `m2.example.net`, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

   You have now tested the connection between `m2.example.net` and `m1.example.net` in both directions.

3. Test the connection from `m3.example.net` to the other two hosts with the following operation set from the `m3.example.net` host, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m2.example.net --port 27017
   ```

If any connection, in any direction fails, check your networking and firewall configuration and reconfigure your environment to allow these connections.

Socket Exceptions when Rebooting More than One Secondary

When you reboot members of a replica set, ensure that the set is able to elect a primary during the maintenance. This means ensuring that a majority of the set's '`votes` (page 663) are available.

When a set's active members can no longer form a majority, the set's *primary* steps down and becomes a *secondary*. The former primary closes all open connections to client applications. Clients attempting to write to the former primary receive socket exceptions and *Connection reset* errors until the set can elect a primary.

Example

Given a three-member replica set where every member has one vote, the set can elect a primary if at least two members can connect to each other. If you reboot the two secondaries at once, the primary steps down and becomes a secondary. Until at least another secondary becomes available, i.e. at least one of the rebooted secondaries also becomes available, the set has no primary and cannot elect a new primary.

For more information on votes, see *Replica Set Elections* (page 583). For related information on connection errors, see *Does TCP keepalive time affect sharded clusters and replica sets?* (page 800).

Check the Size of the Oplog

A larger *oplog* can give a replica set a greater tolerance for lag, and make the set more resilient.

To check the size of the oplog for a given *replica set* member, connect to the member in a `mongo` shell and run the `rs.printReplicationInfo()` method.

The output displays the size of the oplog and the date ranges of the operations contained in the oplog. In the following example, the oplog is about 10MB and is able to fit about 26 hours (94400 seconds) of operations:

```
configured oplog size:   10.10546875MB
log length start to end: 94400 (26.22hrs)
oplog first event time:  Mon Mar 19 2012 13:50:38 GMT-0400 (EDT)
oplog last event time:   Wed Oct 03 2012 14:59:10 GMT-0400 (EDT)
now:                     Wed Oct 03 2012 15:00:21 GMT-0400 (EDT)
```

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations; however, many users prefer to have 72 hours or even a week's work of operations.

For more information on how oplog size affects operations, see:

- *Oplog Size* (page 597),
- *Delayed Replica Set Members* (page 573), and
- *Check the Replication Lag* (page 654).

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 634) tutorial.

Oplog Entry Timestamp Error

Consider the following error in `mongod` output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection.  Terminating mongod after 30
<timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the `ts` field in the last *oplog* entry causes this error. The correct data type is Timestamp.

Check the type of the `ts` value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the `ts` value is a Timestamp. The `$type` operator allows you to select *BSON type* 17, is the Timestamp data type.

If the queries don't return the same document, then the last document in the oplog has the wrong data type in the `ts` field.

Example

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where `ts` has the `Timestamp` type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the `ts` field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
                    { $set: { ts: new Timestamp(1347982456000, 1)}})
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

Duplicate Key Error on `local.slaves`

The *duplicate key on local.slaves* error, occurs when a *secondary* or *slave* changes its hostname and the *primary* or *master* tries to update its `local.slaves` collection with the new name. The update fails because it contains the same `_id` value as the document containing the previous hostname. The error itself will resemble the following.

```
exception: E11000 duplicate key error index: local.slaves.$_id_  dup key: { : ObjectId('<object ID>')
```

This is a benign error and does not affect replication operations on the *secondary* or *slave*.

To prevent the error from appearing, drop the `local.slaves` collection from the *primary* or *master*, with the following sequence of operations in the `mongo` shell:

```
use local
db.slaves.drop()
```

The next time a *secondary* or *slave* polls the *primary* or *master*, the *primary* or *master* recreates the `local.slaves` collection.

9.4 Replication Reference

On this page

9.4.1 Replication Methods in the `mongo` Shell

Name	Description
`rs.add()`	Adds a member to a replica set.
`rs.addArb()`	Adds an *arbiter* to a replica set.
`rs.conf()`	Returns the replica set configuration document.
`rs.freeze()`	Prevents the current member from seeking election as primary for a period of time.
`rs.help()`	Returns basic help text for *replica set* functions.
`rs.initiate()`	Initializes a new replica set.
`rs.printReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the primary.
`rs.printSlaveReplicationInfo()`	Prints a report of the status of the replica set from the perspective of the secondaries.
`rs.reconfig()`	Re-configures a replica set by applying a new replica set configuration object.
`rs.remove()`	Remove a member from a replica set.
`rs.slaveOk()`	Sets the `slaveOk` property for the current connection. Deprecated. Use `readPref()` and `Mongo.setReadPref()` to set *read preference*.
`rs.status()`	Returns a document with information about the state of the replica set.
`rs.stepDown()`	Causes the current *primary* to become a secondary which forces an *election*.
`rs.syncFrom()`	Sets the member that this replica set member will sync from, overriding the default sync target selection logic.

9.4.2 Replication Database Commands

Name	Description
`replSetFreeze`	Prevents the current member from seeking election as *primary* for a period of time.
`replSetGetStatus`	Returns a document that reports on the status of the replica set.
`replSetInitiate`	Initializes a new replica set.
`replSetMaintenance`	Enables or disables a maintenance mode, which puts a *secondary* node in a `RECOVERING` state.
`replSetReconfig`	Applies a new configuration to an existing replica set.
`replSetStepDown`	Forces the current *primary* to *step down* and become a *secondary*, forcing an election.
`replSetSyncFrom`	Explicitly override the default logic for selecting a member to replicate from.
`resync`	Forces a `mongod` to re-synchronize from the *master*. For master-slave replication only.
`applyOps`	Internal command that applies *oplog* entries to the current data set.
`isMaster`	Displays information about this member's role in the replica set, including whether it is the master.
`getoptime`	Internal command to support replication, returns the optime.

9.4.3 Replica Set Reference Documentation

Replica Set Configuration **(page 659)** Complete documentation of the *replica set* configuration object returned by `rs.conf()`.

The local Database **(page 664)** Complete documentation of the content of the `local` database that `mongod` instances use to support replication.

Replica Set Member States **(page 667)** Reference for the replica set member states.

Read Preference Reference **(page 669)** Complete documentation of the five read preference modes that the MongoDB drivers support.

Replica Set Configuration

On this page

The configuration for a replica set is stored as a document in the `system.replset` (page 666) collection in the *local database* (page 664).

Replica Set Configuration Document

The following document provides a representation of a replica set configuration document. The configuration of your replica set may include only a subset of these settings:

```
{
  _id: <string>,
  version: <int>,
  members: [
    {
      _id: <int>,
      host: <string>,
      arbiterOnly: <boolean>,
      buildIndexes: <boolean>,
      hidden: <boolean>,
      priority: <number>,
      tags: <document>,
      slaveDelay: <int>,
      votes: <number>
    },
    ...
  ],
  settings: {
    getLastErrorDefaults : <document>,
    chainingAllowed : <boolean>,
    getLastErrorModes : <document>,
    heartbeatTimeoutSecs: <int>
  }
}
```

Configuration Settings

`local.system.replset.`**`_id`**

Type: string

The name of the replica set. Once set, you cannot change the name of a replica set.

See

`replSetName` or `--replSet` for information on setting the replica set name.

`local.system.replset.`**`version`**

An incrementing number used to distinguish revisions of the replica set configuration object from previous iterations of the configuration.

replset.members
local.system.replset.**members**

Type: array

An array of member configuration documents, one for each member of the replica set. The members (page 661) array is a zero-indexed array.

Each member-specific configuration document can contain the following fields:

local.system.replset.members[n].**_id**

Type: integer

A numeric identifier of every member in the replica set. Once set, you cannot change the _id (page 661) of a member.

Note: When updating the replica configuration object, access the replica set members in the members (page 661) array with the **array index**. The array index begins with 0. Do **not** confuse this index value with the value of the _id (page 661) field in each document in the members (page 661) array.

---

local.system.replset.members[n].**host**

Type: string

The hostname and, if specified, the port number, of the set member.

The hostname name must be resolvable for every host in the replica set.

Warning: host (page 661) 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

A boolean that identifies an arbiter. A value of true indicates that the member is an arbiter.

When using the rs.addArb() method to add an arbiter, the method automatically sets arbiterOnly (page 661) to true for the added member.

local.system.replset.members[n].**buildIndexes**

Optional.

Type: boolean

Default: true

A boolean that indicates whether the mongod builds *indexes* on this member. You can only set this value when adding a member to a replica set. You cannot change buildIndexes (page 661) field after the member has been added to the set. To add a member, see rs.add() and rs.reconfig().

Do not set to false for mongod instances that receive queries from clients.

Setting buildIndexes to false may be useful if **all** the following conditions are true:

- you are only using this instance to perform backups using mongodump, *and*
- this member will receive no queries, *and*
- index creation and maintenance overburdens the host system.

Even if set to `false`, secondaries *will* build indexes on the `_id` field in order to facilitate operations required for replication.

> **Warning:** If you set `buildIndexes` (page 661) to `false`, you must also set `priority` (page 662) to `0`. If `priority` (page 662) is not `0`, MongoDB will return an error when attempting to add a member with `buildIndexes` (page 661) equal to `false`.
> To ensure the member receives no queries, you should make all instances that do not build indexes hidden.
> Other secondaries cannot replicate from a member where `buildIndexes` (page 661) is false.

`local.system.replset.members[n].`**`hidden`**
Optional.

Type: boolean

Default: false

When this value is `true`, the replica set hides this instance and does not include the member in the output of `db.isMaster()` or `isMaster`. This prevents read operations (i.e. queries) from ever reaching this host by way of secondary *read preference*.

See also:

Hidden Replica Set Members (page 572)

`local.system.replset.members[n].`**`priority`**
Optional.

Type: Number, between 0 and 1000.

Default: 1.0

A number that indicates the relative eligibility of a member to become a *primary*.

Specify higher values to make a member *more* eligible to become *primary*, and lower values to make the member *less* eligible. 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 662) of `0` makes it impossible for a member to become primary.

See also:

Replica Set Elections (page 583).

`local.system.replset.members[n].`**`tags`**
Optional.

Type: document

Default: none

A document that contains arbitrary field and value pairs for describing or *tagging* members in order to extend *write concern* (page 135) and *read preference* (page 669) and thereby allowing configurable data center awareness.

Use `tags` to configure write concerns in conjunction with `getLastErrorModes` (page 664) and `getLastErrorDefaults` (page 664).

Important: In tag sets, all tag values must be strings.

For more information on configuring tag sets for read preference and write concern, see *Configure Replica Set Tag Sets* (page 641).

`local.system.replset.members[n].slaveDelay`
Optional.

Type: integer

Default: 0

The number of seconds "behind" the primary that this replica set member should "lag".

Use this option to create *delayed members* (page 573). Delayed members maintain a copy of the data that reflects the state of the data at some time in the past.

See also:

Delayed Replica Set Members (page 573)

`local.system.replset.members[n].votes`
Optional.

Type: integer

Default: 1

The number of votes a server will cast in a *replica set election* (page 583). The number of votes each member has can be either `1` or `0`.

A replica set can have up to 12 members, but can have at most only 7 *voting* members. If you need more than 7 members in one replica set, set `votes` (page 663) to `0` for the additional non-voting members.

Note: Deprecated since version 2.6: `votes` (page 663) values greater than `1`.

Earlier versions of MongoDB allowed a member to have more than `1` vote by setting `votes` (page 663) to a value greater than `1`. Setting `votes` (page 663) to value greater than `1` now produces a warning message.

`replset.settings`
`local.system.replset.settings`
Optional.

Type: document

A document that contains configuration options that apply to the whole replica set.

The `settings` (page 663) document contain the following fields:

`local.system.replset.settings.chainingAllowed`
New in version 2.2.4.

Optional.

Type: boolean

Default: true

When `chainingAllowed` (page 663) is `true`, the replica set allows *secondary* members to replicate from other secondary members. When `chainingAllowed` (page 663) is `false`, secondaries can replicate only from the *primary*.

When you run `rs.conf()` to view a replica set's configuration, the `chainingAllowed` (page 663) field appears only when set to `false`. If not set, `chainingAllowed` (page 663) is `true`.

See also:

Manage Chained Replication (page 647)

`local.system.replset.settings.`**`getLastErrorDefaults`**
Optional.

Type: document

A document that specifies the *write concern* (page 589) for the replica set. The replica set will use this write concern only when *write operations* (page 838) or `getLastError` specify no other write concern.

If `getLastErrorDefaults` (page 664) is not set, the default write concern for the replica set only requires confirmation from the primary.

`local.system.replset.settings.`**`getLastErrorModes`**
Optional.

Type: document

A document used to define an extended *write concern* through the use of `tags` (page 662). The extended *write concern* can provide *data-center awareness*.

For example, the following document defines an extended write concern named `eastCoast` and associates with a write to a member that has the `east` tag.

```
{ getLastErrorModes: { eastCoast: { "east": 1 } } }
```

Write operations to the replica set can use the extended write concern, e.g. `{ w: "eastCoast" }`.

See *Configure Replica Set Tag Sets* (page 641) for more information and example.

`local.system.replset.settings.`**`heartbeatTimeoutSecs`**
Optional.

Type: int

Default: 10

Number of seconds that the replica set members wait for a successful heartbeat from each other. If a member does not respond in time, other members mark the delinquent member as inaccessible.

View Replica Set Configuration

To view the current configuration for a replica set, use the `rs.conf()` method. See `rs.conf()` for more information.

Modify Replica Set Configuration

To modify the configuration for a replica set, use the `rs.reconfig()` method, passing a configuration document to the method. See `rs.reconfig()` for more information.

The `local` Database

On this page

Overview

Every mongod instance has its own local database, which stores data used in the replication process, and other instance-specific data. The local database is invisible to replication: collections in the local database are not replicated.

In replication, the local database store stores internal replication data for each member of a *replica set*. The local stores the following collections:

Changed in version 2.4: When running with authentication (i.e. authorization), authenticating to the local database is **not** equivalent to authenticating to the admin database. In previous versions, authenticating to the local database provided access to all databases.

Collection on all mongod Instances

local.**startup_log**

On startup, each mongod instance inserts a document into startup_log (page 665) with diagnostic information about the mongod instance itself and host information. startup_log (page 665) 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 665) 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 665) 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 666) in the system's local time zone.

`local.startup_log.cmdLine`
An embedded document that reports the `mongod` runtime options and their values.

`local.startup_log.pid`
The process identifier for this process.

`local.startup_log.buildinfo`
An embedded document that reports information about the build environment and settings used to compile this `mongod`. This is the same output as `buildInfo`. See `buildInfo`.

Collections on Replica Set Members

`local.system.replset`
`local.system.replset` (page 666) holds the replica set's configuration object as its single document. To view the object's configuration information, issue `rs.conf()` from the `mongo` shell. You can also query this collection directly.

`local.oplog.rs`
`local.oplog.rs` (page 666) is the capped collection that holds the *oplog*. You set its size at creation using the `oplogSizeMB` setting. To resize the oplog after replica set initiation, use the *Change the Size of the Oplog* (page 634) procedure. For additional information, see the *Oplog Size* (page 597) 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.

Replica Set Member States

On this page

Each member of a replica set has a state that reflects its disposition within the set.

Number	Name	State Description
0	STARTUP (page 668)	Not yet an active member of any set. All members start up in this state. The mongod parses the *replica set configuration document* (page 625) while in STARTUP (page 668).
1	PRIMARY (page 667)	The member in state *primary* (page 568) is the only member that can accept write operations.
2	SECONDARY (page 667)	A member in state *secondary* (page 569) is replicating the data store. Data is available for reads, although they may be stale.
3	RECOVERING (page 668)	A member in this state is replicating the data store but does not yet have a consistent view of the data. Data is not available for reads until the member transitions to state *secondary* (page 569).
5	STARTUP2 (page 668)	The member has joined the set and is running an initial sync.
6	UNKNOWN (page 668)	The member's state, as seen from another member of the set, is not yet known.
7	ARBITER (page 667)	*Arbiters* (page **??**) do not replicate data and exist solely to participate in elections.
8	DOWN (page 668)	The member, as seen from another member of the set, is unreachable.
9	ROLLBACK (page 668)	This member is actively performing a *rollback* (page 587). Data is not available for reads.
10	REMOVED (page 668)	This member was once in a replica set but was subsequently removed.

States

Core States

PRIMARY
: Members in PRIMARY (page 667) state accept write operations. A replica set has at most one primary at a time. A SECONDARY (page 667) member becomes primary after an *election* (page 583). Members in the PRIMARY (page 667) state are eligible to vote.

SECONDARY
: Members in SECONDARY (page 667) 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 667) state if the primary becomes unavailable.

ARBITER

Members in ARBITER (page 667) 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 667) state if the set would otherwise have an even number of members, and could suffer from tied elections. There should only be at most one arbiter configured in any replica set.

See *Replica Set Members* (page 567) for more information on core states.

Other States

STARTUP

Each member of a replica set starts up in STARTUP (page 668) state. mongod then loads that member's replica set configuration, and transitions the member's state to STARTUP2 (page 668). Members in STARTUP (page 668) are not eligible to vote, as they are not yet a recognized member of any replica set.

STARTUP2

Each member of a replica set enters the STARTUP2 (page 668) state as soon as mongod finishes loading that member's configuration, at which time it becomes an active member of the replica set. The member then decides whether or not to undertake an initial sync. If a member begins an initial sync, the member remains in STARTUP2 (page 668) until all data is copied and all indexes are built. Afterwards, the member transitions to RECOVERING (page 668).

RECOVERING

A member of a replica set enters RECOVERING (page 668) state when it is not ready to accept reads. The RECOVERING (page 668) state can occur during normal operation, and doesn't necessarily reflect an error condition. Members in the RECOVERING (page 668) state are eligible to vote in elections, but are not eligible to enter the PRIMARY (page 667) state.

A member transitions from RECOVERING (page 668) to SECONDARY (page 667) after replicating enough data to guarantee a consistent view of the data for client reads. The only difference between RECOVERING (page 668) and SECONDARY (page 667) states is that RECOVERING (page 668) prohibits client reads and SECONDARY (page 667) permits them. SECONDARY (page 667) state does not guarantee anything about the staleness of the data with respect to the primary.

Due to overload, a *secondary* may fall far enough behind the other members of the replica set such that it may need to *resync* (page 640) with the rest of the set. When this happens, the member enters the RECOVERING (page 668) state and requires manual intervention.

Members in SECONDARY (page 667) state can be forced into state RECOVERING (page 668) to prevent client reads by using maintenance mode. You can toggle maintenance mode with the replSetMaintenance command. Maintenance mode is also enabled automatically by the compact and touch commands. While maintenance mode is on, a member will remain in RECOVERING (page 668) state and will reject client reads.

UNKNOWN

Members that have never communicated status information to the replica set are in the UNKNOWN (page 668) state.

DOWN

Members that lose their connection to the replica set are seen as DOWN (page 668) by the remaining members of the set.

REMOVED

Members that are removed from the replica set enter the REMOVED (page 668) state. When members enter the REMOVED (page 668) state, the logs will mark this event with a replSet REMOVED message entry.

ROLLBACK

Whenever the replica set replaces a *primary* in an election, the old primary may contain documents that did not replicate to the *secondary* members. In this case, the old primary member reverts those writes. During *rollback* (page 587), the member will have ROLLBACK (page 668) state.

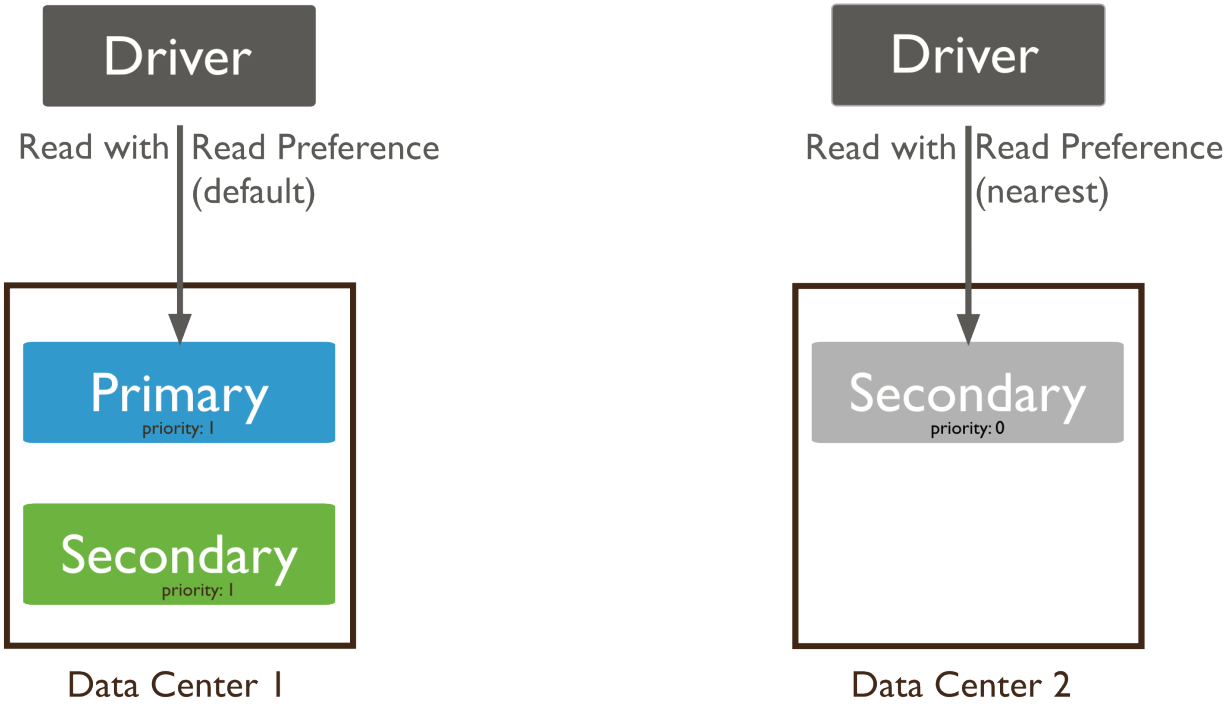
FATAL

A member in `FATAL` (page 668) encountered an unrecoverable error. The member must be shut down and restarted; a resync may be required as well.

Read Preference Reference

On this page

Read preference describes how MongoDB clients route read operations to the members of a *replica set*.



By default, an application directs its read operations to the *primary* member in a *replica set*. Because write operations are issued to the single primary, reading from the primary returns the latest version of a document [20].

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.

[20] In some circumstances, two nodes in a replica set may *transiently* believe that they are the primary, but at most, one of them will be able to complete writes with *{w: majority} write concern* (page 135). The node that can complete *{w: majority}* (page 135) writes is the current primary, and the other node is a former primary that has not yet recognized its demotion, typically due to a *network partition*. When this occurs, clients that connect to the former primary may observe stale data despite having requested read preference `primary` (page 670).

Read Preference Mode	Description
primary (page 670)	Default mode. All operations read from the current replica set *primary*.
primaryPreferred (page 670)	In most situations, operations read from the *primary* but if it is unavailable, operations read from *secondary* members.
secondary (page 670)	All operations read from the *secondary* members of the replica set.
secondaryPreferred (page 671)	In most situations, operations read from *secondary* members but if no *secondary* members are available, operations read from the *primary*.
nearest (page 671)	Operations read from member of the *replica set* with the least network latency, irrespective of the member's type.

Note: The read preference does not affect the visibility of data; i.e, clients can see the results of writes before they are made durable:

- Regardless of *write concern* (page 135), other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
- Clients can read data which may be subsequently *rolled back* (page 587).

Read Preference Modes

primary

All read operations use only the current replica set *primary*. [5] This is the default read mode. If the primary is unavailable, read operations produce an error or throw an exception.

The primary (page 670) read preference mode is not compatible with read preference modes that use *tag sets* (page 594). If you specify a tag set with primary (page 670), 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 594), 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 670) mode may return stale data in some situations.

> **Warning:** Changed in version 2.2: mongos added full support for read preferences.
> When connecting to a mongos instance older than 2.2, using a client that supports read preference modes, primaryPreferred (page 670) 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 594), 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 594). If no secondaries have matching tags, the read operation produces an error. [21]

[21] If your set has more than one secondary, and you use the secondary (page 670) read preference mode, consider the following effect. If

Read operations using the secondary (page 670) 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 594), 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 594). If no secondaries have matching tags, the client ignores tags and reads from the primary.

Read operations using the secondaryPreferred (page 671) mode may return stale data.

nearest

The driver reads from the *nearest* member of the *set* according to the *member selection* (page 594) process. Reads in the nearest (page 671) mode do not consider the member's *type*. Reads in nearest (page 671) 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 594), 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 594).

Read operations using the nearest (page 671) 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 671) mode prefers low latency reads over a member's *primary* or *secondary* status.

For nearest (page 671), 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 594) for more information.

Use Cases

Depending on the requirements of an application, you can configure different applications to use different read preferences, or use different read preferences for different queries in the same application. Consider the following applications for different read preference strategies.

Maximize Consistency To avoid *stale* reads, use primary (page 670) read preference. If the primary is unavailable, e.g. during elections or when a majority of the replica set is not accessible, read operations produce an error or throw an exception.

In some rare edge cases, it may be possible for a replica set to temporarily have two primaries. For example,

- A partial *network partition* may segregate a primary ($\mathtt{p}_{old}$) into a partition with a minority of the nodes, while the other side of the partition contains a majority of nodes. The partition with the majority will elect a new primary ($\mathtt{P}_{new}$), but for a brief period, the old primary ($\mathtt{p}_{old}$) may still continue to serve reads and writes, as it has not yet detected that it can only see a minority of nodes in the replica set. During this period, if the old primary ($\mathtt{p}_{old}$) is still visible to clients as a primary, reads from this primary may reflect stale data.
- A primary ($\mathtt{p}_{old}$) may become unresponsive, which will trigger an election and a new primary ($\mathtt{P}_{new}$) can be elected, serving reads and writes. If the unresponsive primary ($\mathtt{p}_{old}$) starts responding again, two primaries will

you have a *three member replica set* (page 577) with a primary and two secondaries, and one secondary becomes unavailable, all secondary (page 670) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.

be visible for a brief period. The brief period will end when p_{old} steps down. However, during the brief period, clients might read from the old primary p_{old}, which can provide stale data.

To increase consistency, you can disable automatic *failover*; however, disabling automatic failover sacrifices availability.

Maximize Availability To permit read operations when possible, use `primaryPreferred` (page 670). When there's a primary you will get consistent reads [5], but if there is no primary you can still query *secondaries*. However, when using this read mode, consider the situation described in *Reduce load on the primary* (page 672).

Minimize Latency To always read from a low-latency node, use `nearest` (page 671). The driver or `mongos` will read from the nearest member and those no more than 15 milliseconds [22] further away than the nearest member.

`nearest` (page 671) does *not* guarantee consistency. If the nearest member to your application server is a secondary with some replication lag, queries could return stale data. `nearest` (page 671) only reflects network distance and does not reflect I/O or CPU load.

Query From Geographically Distributed Members If the members of a replica set are geographically distributed, you can create replica tags based that reflect the location of the instance and then configure your application to query the members nearby.

For example, if members in "east" and "west" data centers are *tagged* (page 641) `{'dc': 'east'}` and `{'dc': 'west'}`, your application servers in the east data center can read from nearby members with the following read preference:

```
db.collection.find().readPref( { mode: 'nearest',
                                 tags: [ {'dc': 'east'} ] } )
```

Although `nearest` (page 671) already favors members with low network latency, including the tag makes the choice more predictable.

Reduce load on the primary To shift read load from the primary, use mode `secondary` (page 670). Although `secondaryPreferred` (page 671) is tempting for this use case, it carries some 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. If the primary is unable to handle this load, queries will compete with writes. For this reason, use `secondary` (page 670) to distribute read load to replica sets, not `secondaryPreferred` (page 671).

Read Preferences for Database Commands

Because some *database commands* read and return data from the database, all of the official drivers support full *read preference mode semantics* (page 670) for the following commands:

- `group`
- `mapReduce` [23]
- `aggregate` [24]
- `collStats`
- `dbStats`

[22] This threshold is configurable. See `localPingThresholdMs` for `mongos` or your driver documentation for the appropriate setting.

[23] Only "inline" `mapReduce` operations that do not write data support read preference, otherwise these operations must run on the *primary* members.

[24] Using the `$out` pipeline operator forces the aggregation pipeline to run on the primary.

- `count`
- `distinct`
- `geoNear`
- `geoSearch`
- `geoWalk`
- `parallelCollectionScan`

New in version 2.4: `mongos` adds support for routing commands to shards using read preferences. Previously `mongos` sent all commands to shards' primaries.

CHAPTER 10

Sharding

Sharding is the process of storing data records across multiple machines and is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput. Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

Sharding Introduction **(page 675)** A high-level introduction to horizontal scaling, data partitioning, and sharded clusters in MongoDB.

Sharding Concepts **(page 681)** The core documentation of sharded cluster features, configuration, architecture and behavior.

- *Sharded Cluster Components* **(page 681)** A sharded cluster consists of shards, config servers, and `mongos` instances.
- *Sharded Cluster Architectures* **(page 685)** Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.
- *Sharded Cluster Behavior* **(page 687)** Discusses the operations of sharded clusters with regards to the automatic balancing of data in a cluster and other related availability and security considerations.
- *Sharding Mechanics* **(page 697)** Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

Sharded Cluster Tutorials **(page 704)** Tutorials that describe common procedures and administrative operations relevant to the use and maintenance of sharded clusters.

Sharding Reference **(page 753)** Reference for sharding-related functions and operations.

10.1 Sharding Introduction

On this page

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.

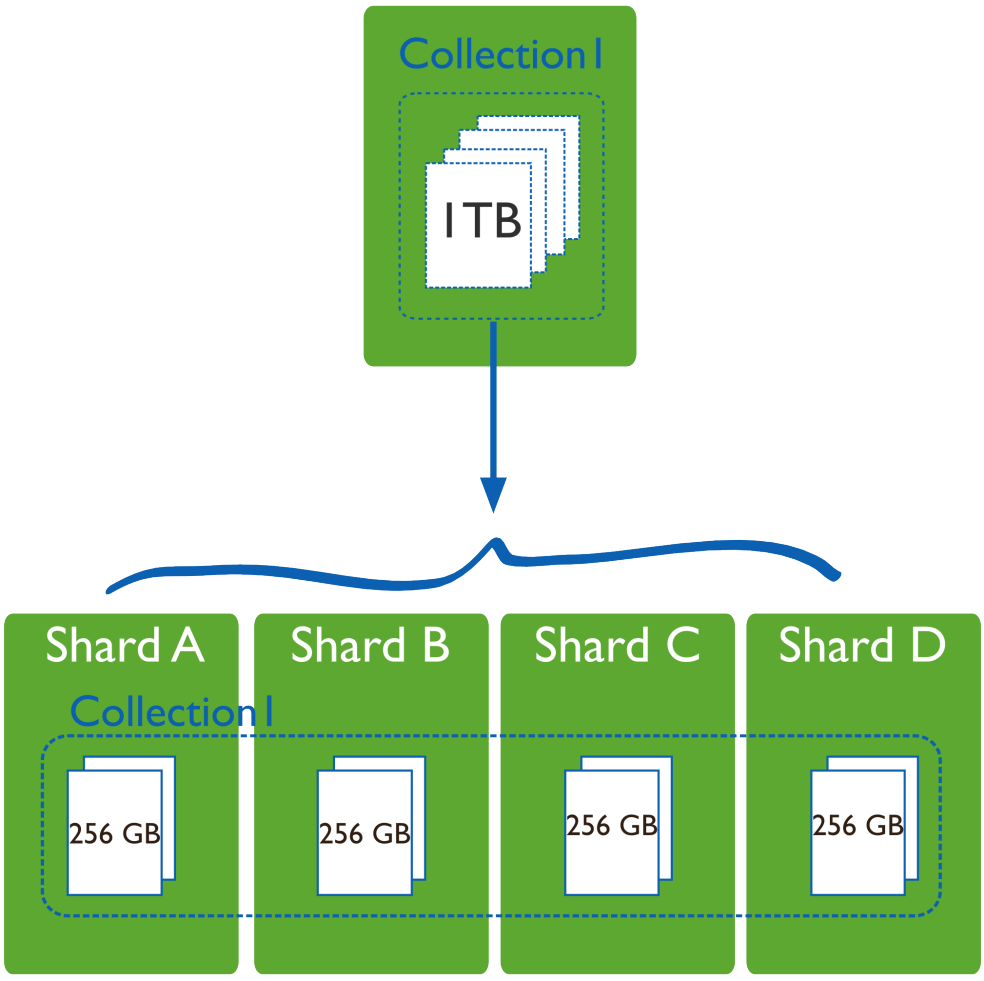
10.1.1 Purpose of Sharding

Database systems with large data sets and high throughput applications can challenge the capacity of a single server. High query rates can exhaust the CPU capacity of the server. Larger data sets exceed the storage capacity of a single machine. Finally, working set sizes larger than the system's RAM stress the I/O capacity of disk drives.

To address these issues of scales, database systems have two basic approaches: **vertical scaling** and **sharding**.

Vertical scaling adds more CPU and storage resources to increase capacity. Scaling by adding capacity has limitations: high performance systems with large numbers of CPUs and large amount of RAM are disproportionately *more expensive* than smaller systems. Additionally, cloud-based providers may only allow users to provision smaller instances. As a result there is a *practical maximum* capability for vertical scaling.

Sharding, or *horizontal scaling*, by contrast, divides the data set and distributes the data over multiple servers, or **shards**. Each shard is an independent database, and collectively, the shards make up a single logical database.



Sharding addresses the challenge of scaling to support high throughput and large data sets:

- Sharding reduces the number of operations each shard handles. Each shard processes fewer operations as the cluster grows. As a result, a cluster can increase capacity and throughput *horizontally*.

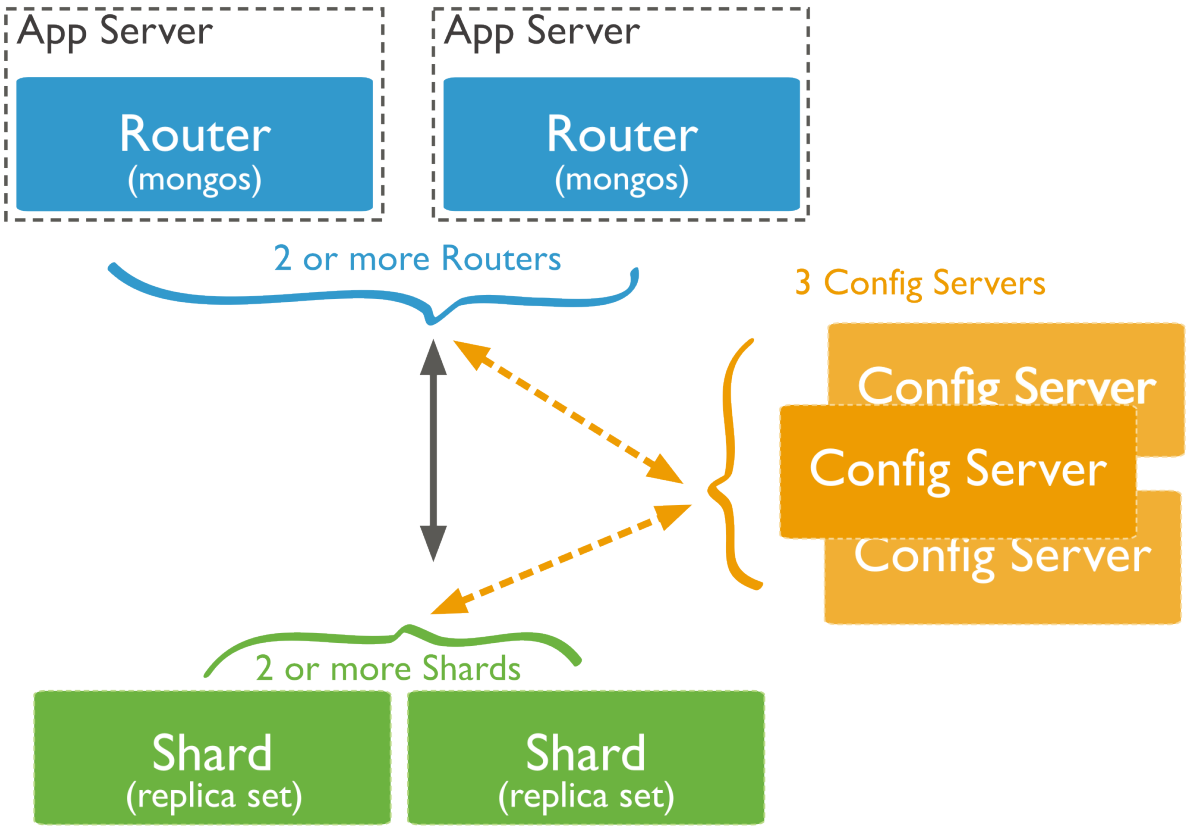
  For example, to insert data, the application only needs to access the shard responsible for that record.

- 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.

10.1.2 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*.

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 567).

Query Routers, or `mongos` instances, interface with client applications and direct operations to the appropriate shard or shards. The query router processes and targets operations to shards and then returns results to the clients. A sharded cluster can contain more than one query router to divide the client request load. A client sends requests to one query router. Most sharded clusters 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.

10.1.3 Data Partitioning

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data by the **shard key**.

[1] For development and testing purposes only, each **shard** can be a single `mongod` instead of a replica set. Do **not** deploy production clusters without 3 config servers.

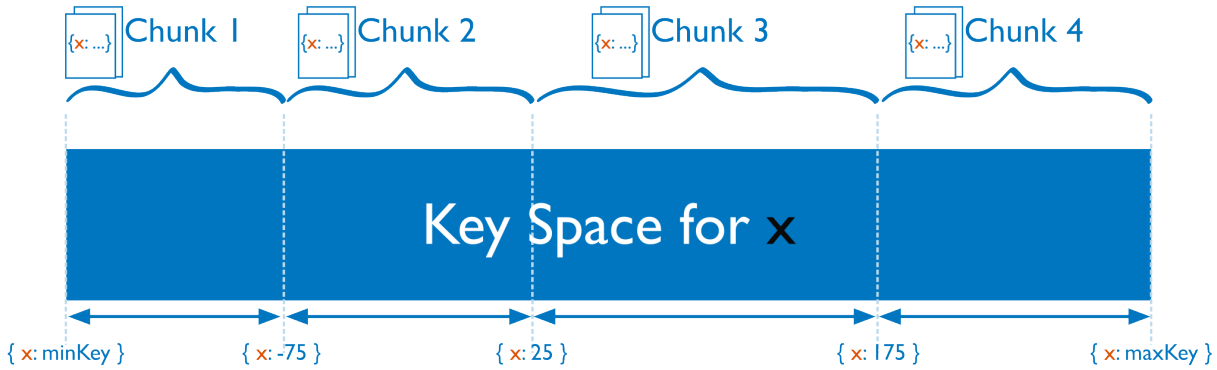
Shard Keys

To shard a collection, you need to select a **shard key**. A *shard key* is either an indexed field or an indexed compound field that exists in every document in the collection. MongoDB divides the shard key values into **chunks** and distributes the *chunks* evenly across the shards. To divide the shard key values into chunks, MongoDB uses either **range based partitioning** or **hash based partitioning**. See the *Shard Key* (page 687) documentation for more information.

Range Based Sharding

For *range-based sharding*, MongoDB divides the data set into ranges determined by the shard key values to provide **range based partitioning**. Consider a numeric shard key: If you visualize a number line that goes from negative infinity to positive infinity, each value of the shard key falls at some point on that line. MongoDB partitions this line into smaller, non-overlapping ranges called **chunks** where a chunk is range of values from some minimum value to some maximum value.

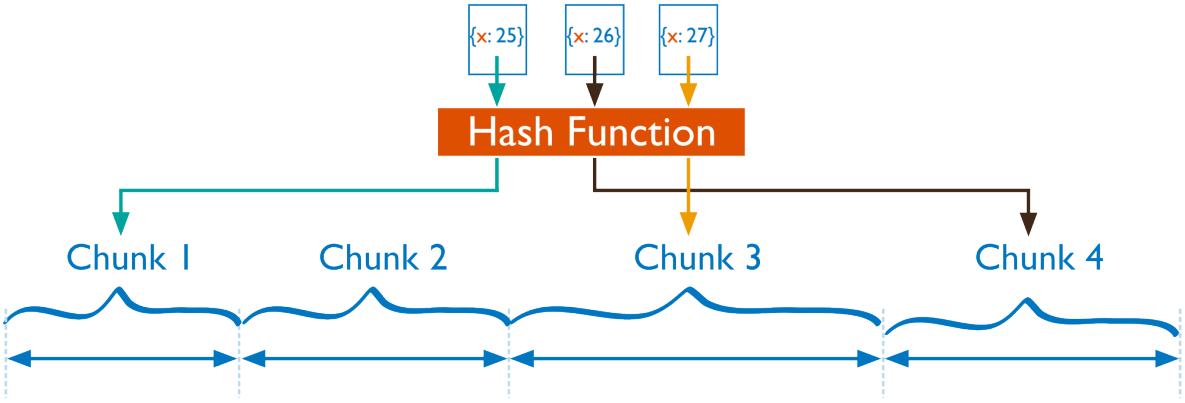
Given a range based partitioning system, documents with "close" shard key values are likely to be in the same chunk, and therefore on the same shard.



Hash Based Sharding

For *hash based partitioning*, MongoDB computes a hash of a field's value, and then uses these hashes to create chunks.

With hash based partitioning, two documents with "close" shard key values are *unlikely* to be part of the same chunk. This ensures a more random distribution of a collection in the cluster.



Performance Distinctions between Range and Hash Based Partitioning

Range based partitioning supports more efficient range queries. Given a range query on the shard key, the query router can easily determine which chunks overlap that range and route the query to only those shards that contain these chunks.

However, range based partitioning can result in an uneven distribution of data, which may negate some of the benefits of sharding. For example, if the shard key is a linearly increasing field, such as time, then all requests for a given time range will map to the same chunk, and thus the same shard. In this situation, a small set of shards may receive the majority of requests and the system would not scale very well.

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.

Customized Data Distribution with Tag Aware Sharding

MongoDB allows administrators to direct the balancing policy using **tag aware sharding**. Administrators create and associate tags with ranges of the shard key, and then assign those tags to the shards. Then, the balancer migrates tagged data to the appropriate shards and ensures that the cluster always enforces the distribution of data that the tags describe.

Tags are the primary mechanism to control the behavior of the balancer and the distribution of chunks in a cluster. Most commonly, tag aware sharding serves to improve the locality of data for sharded clusters that span multiple data centers.

See *Tag Aware Sharding* (page 746) for more information.

10.1.4 Maintaining a Balanced Data Distribution

The addition of new data or the addition of new servers can result in data distribution imbalances within the cluster, such as a particular shard contains significantly more chunks than another shard or a size of a chunk is significantly greater than other chunk sizes.

MongoDB ensures a balanced cluster using two background process: splitting and the balancer.

Splitting

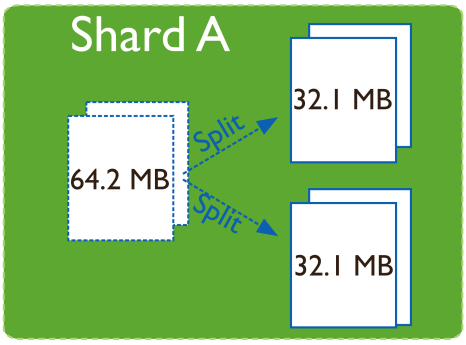
Splitting is a background process that keeps chunks from growing too large. When a chunk grows beyond a *specified chunk size* (page 702), MongoDB splits the chunk in half. Inserts and updates triggers splits. Splits are an efficient meta-data change. To create splits, MongoDB does *not* migrate any data or affect the shards.

Balancing

The *balancer* (page 698) is a background process that manages chunk migrations. The balancer can run from any of the query routers in a cluster.

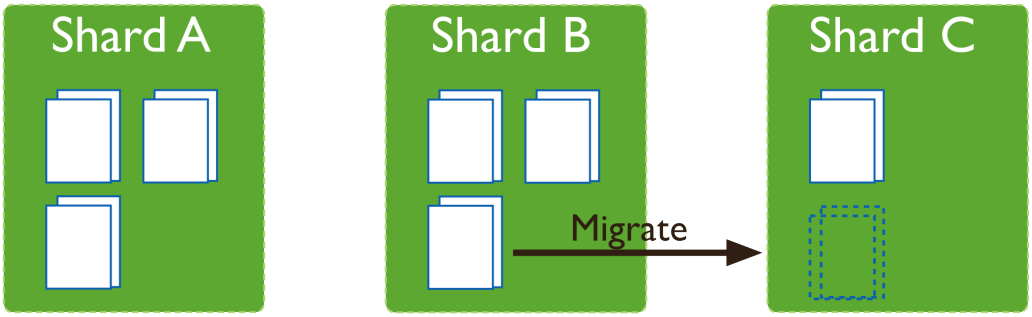
When the distribution of a sharded collection in a cluster is uneven, the balancer process migrates chunks from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances. For example: if collection `users` has 100 chunks on *shard 1* and 50 chunks on *shard 2*, the balancer will migrate chunks from *shard 1* to *shard 2* until the collection achieves balance.

The shards manage *chunk migrations* as a background operation between an *origin shard* and a *destination shard*. During a chunk migration, the *destination shard* is sent all the current documents in the chunk from the *origin shard*.



Next, the destination shard captures and applies all changes made to the data during the migration process. Finally, the metadata regarding the location of the chunk on *config server* is updated.

If there's an error during the migration, the balancer aborts the process leaving the chunk unchanged on the origin shard. MongoDB removes the chunk's data from the origin shard **after** the migration completes successfully.



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 a shard to other shards. After migrating all data and updating the meta data, you can safely remove the shard.

10.1.5 Additional Resources

- Sharding Methods for MongoDB (Presentation)[2]
- Everything You Need to Know About Sharding (Presentation)[3]
- MongoDB for Time Series Data: Sharding[4]
- MongoDB Operations Best Practices White Paper[5]

[2] http://www.mongodb.com/presentations/webinar-sharding-methods-mongodb?jmp=docs
[3] http://www.mongodb.com/presentations/webinar-everything-you-need-know-about-sharding?jmp=docs
[4] http://www.mongodb.com/presentations/mongodb-time-series-data-part-3-sharding?jmp=docs
[5] http://www.mongodb.com/lp/white-paper/ops-best-practices?jmp=docs

- Talk to a MongoDB Expert About Scaling[6]
- MongoDB Deployment Topology Consulting Package[7]

10.2 Sharding Concepts

These documents present the details of sharding in MongoDB. These include the components, the architectures, and the behaviors of MongoDB sharded clusters. For an overview of sharding and sharded clusters, see *Sharding Introduction* (page 675).

Sharded Cluster Components **(page 681)** A sharded cluster consists of shards, config servers, and `mongos` instances.

Shards **(page 682)** A shard is a single server or replica set that holds a part of the sharded collection.

Config Servers **(page 684)** Config servers hold the metadata about the cluster, such as the shard location of the data.

Sharded Cluster Architectures **(page 685)** Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.

Sharded Cluster Requirements **(page 685)** Discusses the requirements for sharded clusters in MongoDB.

Production Cluster Architecture **(page 686)** Outlines the components required to deploy a redundant and highly available sharded cluster.

Continue reading from *Sharded Cluster Architectures* (page 685) for additional descriptions of sharded cluster deployments.

Sharded Cluster Behavior **(page 687)** Discusses the operations of sharded clusters with regards to the automatic balancing of data in a cluster and other related availability and security considerations.

Shard Keys **(page 687)** MongoDB uses the shard key to divide a collection's data across the cluster's shards.

Sharded Cluster High Availability **(page 691)** Sharded clusters provide ways to address some availability concerns.

Sharded Cluster Query Routing **(page 692)** The cluster's routers, or `mongos` instances, send reads and writes to the relevant shard or shards.

Sharding Mechanics **(page 697)** Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

Sharded Collection Balancing **(page 698)** Balancing distributes a sharded collection's data cluster to all of the shards.

Sharded Cluster Metadata **(page 703)** The cluster maintains internal metadata that reflects the location of data within the cluster.

Continue reading from *Sharding Mechanics* (page 697) for more documentation of the behavior and operation of sharded clusters.

10.2.1 Sharded Cluster Components

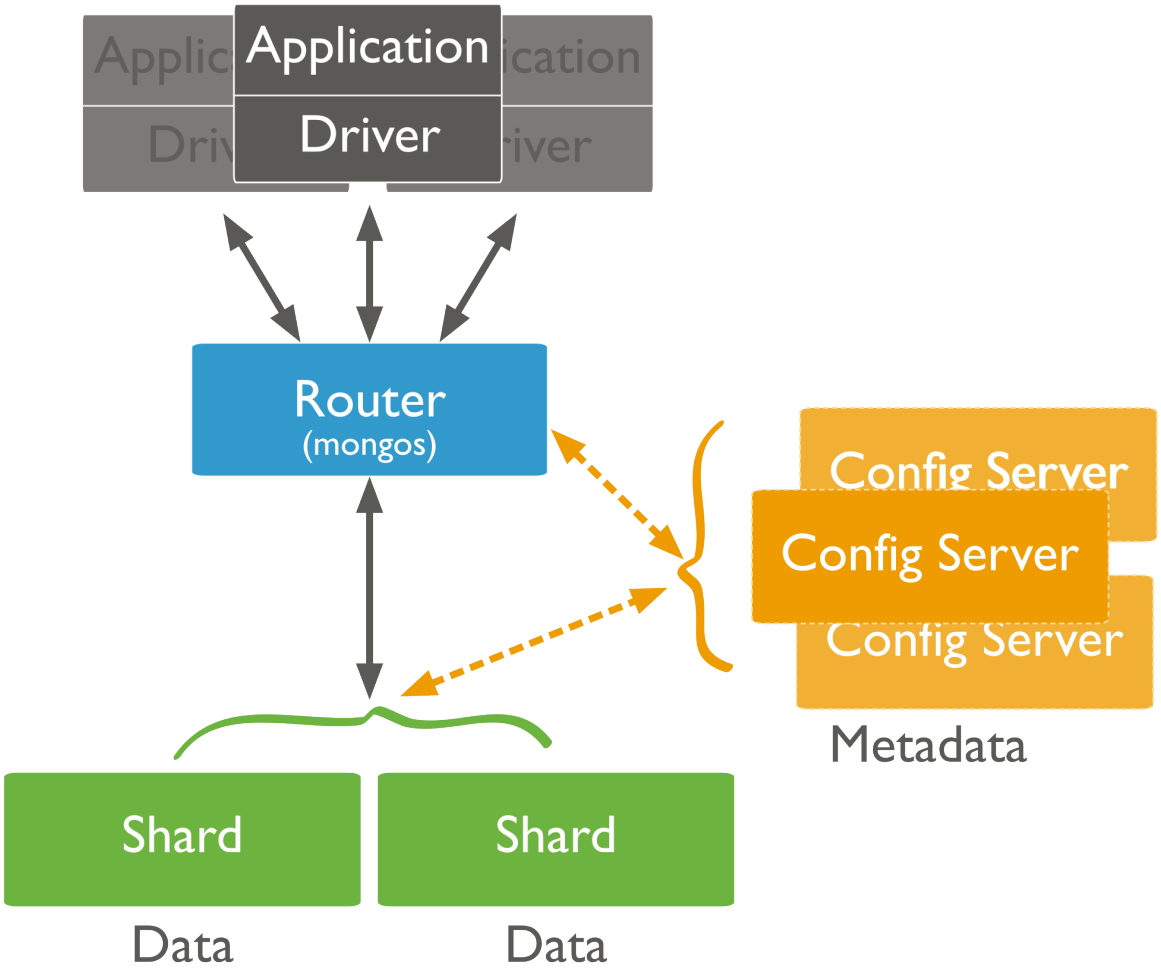
Sharded clusters implement *sharding*. A sharded cluster consists of the following components:

[6] http://www.mongodb.com/lp/contact/planning-for-scale?jmp=docs
[7] https://www.mongodb.com/products/consulting#deployment_topology?jmp=docs

Shards A shard is a MongoDB instance that holds a subset of a collection's data. Each shard is either a single `mongod` instance or a *replica set*. In production, all shards are replica sets. For more information see *Shards* (page 682).

Config Servers Each *config server* (page 684) is a `mongod` instance that holds metadata about the cluster. The metadata maps *chunks* to shards. For more information, see *Config Servers* (page 684).

Routing Instances Each router is a `mongos` instance that routes the reads and writes from applications to the shards. Applications do not access the shards directly. For more information see *Sharded Cluster Query Routing* (page 692).



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 705).

Shards

On this page

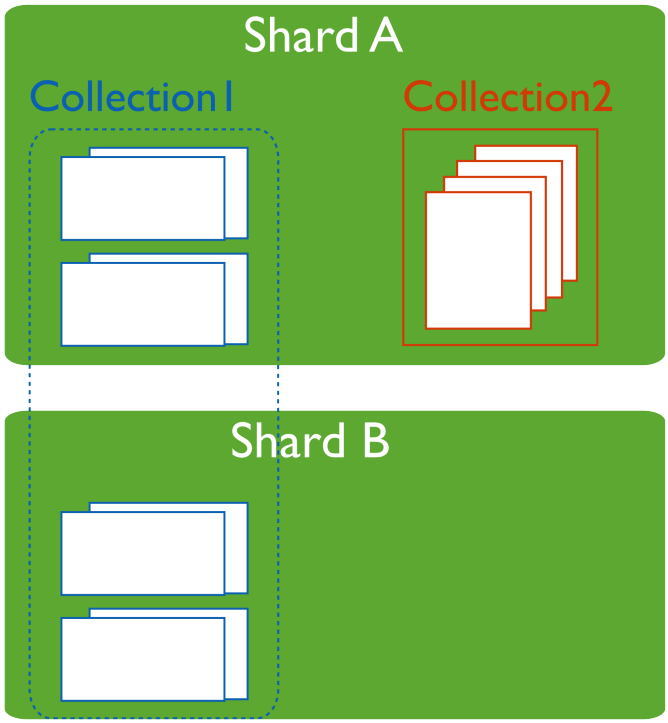
A shard is a *replica set* or a single `mongod` that contains a subset of the data for the sharded cluster. Together, the cluster's shards hold the entire data set for the cluster.

Typically each shard is a replica set. The replica set provides redundancy and high availability for the data in each shard.

Important: MongoDB shards data on a *per collection* basis. You *must* access all data in a sharded cluster via the `mongos` instances. If you connect directly to a shard, you will see only its fraction of the cluster's data. There is no 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" [8] shard that holds all the un-sharded collections in that database.



To change the primary shard for a database, use the `movePrimary` command. The process of migrating the primary shard may take significant time to complete, and you should not access the collections until it completes.

When you deploy a new *sharded cluster* with shards that were previously used as replica sets, all existing databases continue to reside on their original shard. Databases created subsequently may reside on any shard in the cluster.

[8] The term "primary" shard has nothing to do with the term *primary* in the context of *replica sets*.

Shard Status

Use the `sh.status()` method in the `mongo` shell to see an overview of the cluster. This reports includes which shard is primary for the database and the *chunk* distribution across the shards. See `sh.status()` method for more details.

Config Servers

On this page

Config servers are special `mongod` instances that store the *metadata* (page 703) for a sharded cluster.

A production sharded cluster has *exactly three* config servers. All config servers must be available to deploy a sharded cluster or to make any changes to cluster metadata. Config servers *do not* run as replica sets.

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.

Each sharded cluster must have its own config servers. Do not use the same config servers for different sharded clusters.

Tip

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

Read and Write Operations on Config Servers

Config servers store the cluster's metadata in the *config database* (page 754). The `mongos` instances cache this data and use it to route reads and writes to shards.

MongoDB only writes data to the config server when the metadata changes, such as

- after a *chunk migration* (page 700), or
- after a *chunk split* (page 702).

When writing to the three config servers, a coordinator dispatches the same write commands to the three config servers and collects the results. Differing results indicate an inconsistent writes to the config servers and may require manual intervention. Once the config servers become inconsistent, the balancer will not perform any chunk migration and `mongos` will not perform auto-splits of chunks.

MongoDB reads data from the config server in the following cases:

- A new `mongos` starts for the first time, or an existing `mongos` restarts.
- After change in the cluster metadata, such as after a chunk migration.

MongoDB also uses the config server to manage distributed locks.

Config Server Availability

If one or two config servers become unavailable, the cluster's metadata becomes *read only*. You can still read and write data from the shards, but no chunk migrations or splits will occur until all three servers are available.

If all three config servers are unavailable, you can still use the cluster if you do not restart the `mongos` instances until after the config servers are accessible again. If you restart the `mongos` instances before the config servers are available, the `mongos` will be unable to route reads and writes.

Clusters become inoperable without the cluster metadata. To ensure that the config servers remain available and intact, backups of config servers are critical. The data on the config server is small compared to the data stored in a cluster, and the config server has a relatively low activity load. These properties facilitate finding a window to back up the config servers.

If the name or address that a sharded cluster uses to connect to a config server changes, you must restart **every** `mongod` and `mongos` instance in the sharded cluster. Avoid downtime by using CNAMEs to identify config servers within the MongoDB deployment.

See *Renaming Config Servers and Cluster Availability* (page 692) for more information.

10.2.2 Sharded Cluster Architectures

The following documents introduce deployment patterns for sharded clusters.

Sharded Cluster Requirements

On this page

While sharding is a powerful and compelling feature, sharded clusters have significant infrastructure requirements and increases the overall complexity of a deployment. As a result, only deploy sharded clusters when indicated by application and operational requirements

Sharding is the *only* solution for some classes of deployments. Use *sharded clusters* if:

- your data set approaches or exceeds the storage capacity of a single MongoDB instance.
- the size of your system's active *working set will soon* exceed the capacity of your system's *maximum* RAM.
- a single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add complexity to your system without adding much benefit.

Important: It takes time and resources to deploy sharding. If your system has *already* reached or exceeded its capacity, it will be difficult to deploy sharding without impacting your application.

As a result, if you think you will need to partition your database in the future, **do not** wait until your system is over capacity 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 698) will not begin moving data across shards until the imbalance of chunks among the shards exceeds the *migration threshold* (page 699). 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 702) for more information.

Production Cluster Architecture

In a production cluster, you must ensure that data is redundant and that your systems are highly available. To that end, a production cluster must have the following components:

- **Three Config Servers** Each *config server* (page 684) must be on separate machines. A single *sharded cluster* must have exclusive use of its *config servers* (page 684). If you have multiple sharded clusters, you will need to have a group of config servers for each cluster.
- **Two or More Replica Sets As Shards** These replica sets are the *shards*. For information on replica sets, see *Replication* (page 563).
- **One or More Query Routers (`mongos`)** The `mongos` instances are the routers for the cluster. Typically, deployments have one `mongos` instance on each application server.

  You may also deploy a group of `mongos` instances and use a proxy/load balancer between the application and the `mongos`. In these deployments, you *must* configure the load balancer for *client affinity* so that every connection from a single client reaches the same `mongos`.

  Because cursors and other resources are specific to an single `mongos` instance, each client must interact with only one `mongos` instance.

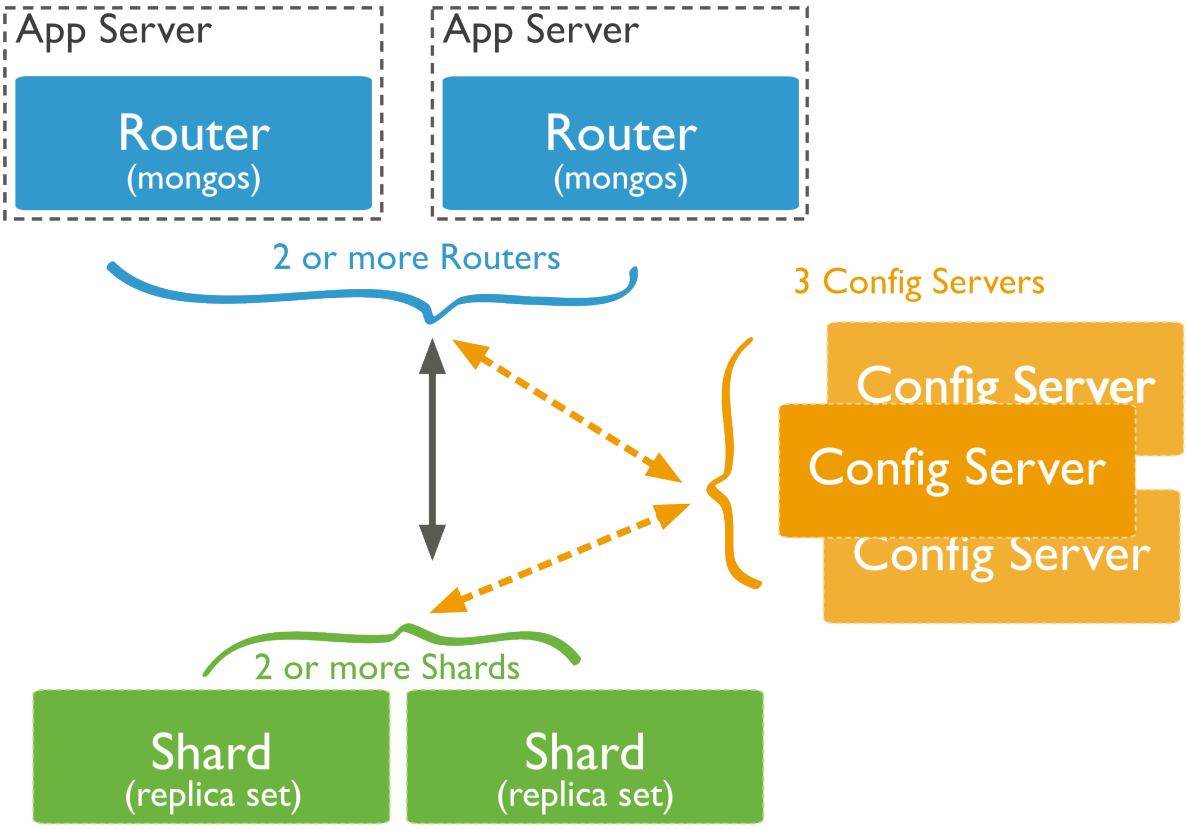
See also:

Deploy a Sharded Cluster (page 705)

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 684).
- At least one shard. Shards are either *replica sets* or a standalone `mongod` instances.
- One `mongos` instance.

See

Production Cluster Architecture (page 686)

10.2.3 Sharded Cluster Behavior

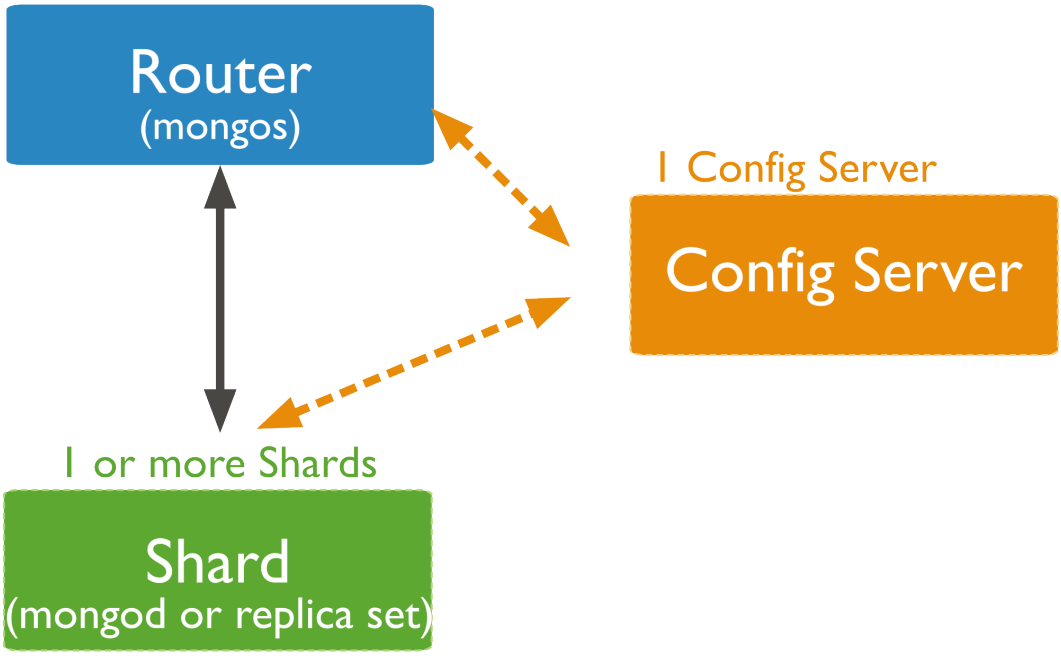
These documents address the distribution of data and queries to a sharded cluster as well as specific security and availability considerations for sharded clusters.

Shard Keys **(page 687)** MongoDB uses the shard key to divide a collection's data across the cluster's shards.

Sharded Cluster High Availability **(page 691)** Sharded clusters provide ways to address some availability concerns.

Sharded Cluster Query Routing **(page 692)** The cluster's routers, or `mongos` instances, send reads and writes to the relevant shard or shards.

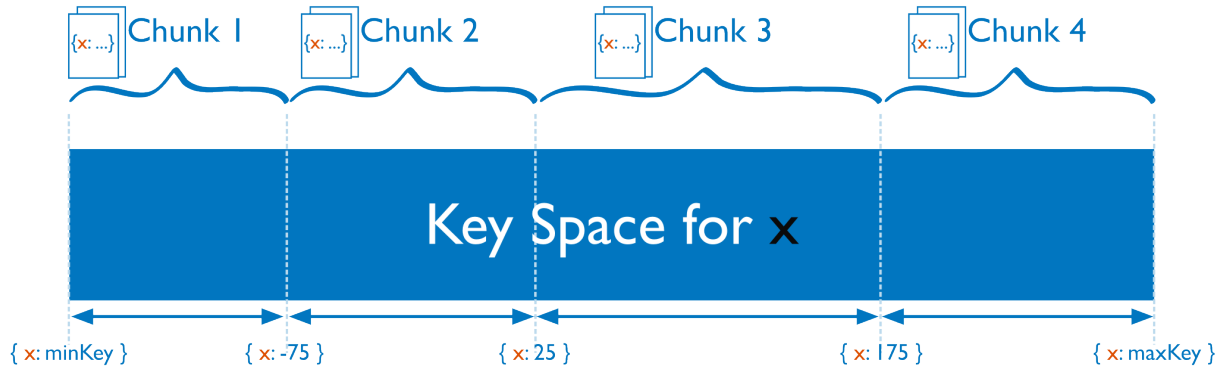
Shard Keys



On this page

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.



When a chunk grows beyond the *chunk size* (page 702), MongoDB attempts to *split* the chunk into smaller chunks, always based on ranges in the shard key.

Considerations

Shard keys are immutable and cannot be changed after insertion. See the *system limits for sharded cluster* for more information.

The index on the shard key **cannot** be a *multikey index* (page 491).

Hashed Shard Keys

New in version 2.4.

Hashed shard keys use a *hashed index* (page 524) of a single field as the *shard key* to partition data across your sharded cluster.

The field you choose as your hashed shard key should have a good cardinality, or large number of different values. Hashed keys work well with fields that increase monotonically like *ObjectId* values or timestamps.

If you shard an empty collection using a hashed shard key, MongoDB will automatically create and migrate chunks so that each shard has two chunks. You can control how many chunks MongoDB will create with the `numInitialChunks` parameter to `shardCollection` or by manually creating chunks on the empty collection using the `split` command.

To shard a collection using a hashed shard key, see *Shard a Collection Using a Hashed Shard Key* (page 711).

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

Impacts of Shard Keys on Cluster Operations

The shard key affects write and query performance by determining how the MongoDB partitions data in the cluster and how effectively the `mongos` instances can direct operations to the cluster. Consider the following operational impacts of shard key selection:

Write Scaling Some possible shard keys will allow your application to take advantage of the increased write capacity that the cluster can provide, while others do not. Consider the following example where you shard by the values of the default *_id* field, which is *ObjectId*.

MongoDB generates `ObjectId` values upon document creation to produce a unique identifier for the object. However, the most significant bits of data in this value represent a time stamp, which means that they increment in a regular and predictable pattern. Even though this value has *high cardinality* (page 710), when using this, *any date, or other monotonically increasing number* as the shard key, all insert operations will be storing data into a single chunk, and therefore, a single shard. As a result, the write capacity of this shard will define the effective write capacity of the cluster.

A shard key that increases monotonically will not hinder performance if you have a very low insert rate, or if most of your write operations are `update()` operations distributed through your entire data set. Generally, choose shard keys that have *both* high cardinality and will distribute write operations across the *entire cluster*.

Typically, a computed shard key that has some amount of "randomness," such as ones that include a cryptographic hash (i.e. MD5 or SHA1) of other content in the document, will allow the cluster to scale write operations. However, random shard keys do not typically provide *query isolation* (page 690), which is another important characteristic of shard keys.

New in version 2.4: MongoDB makes it possible to shard a collection on a hashed index. This can greatly improve write scaling. See *Shard a Collection Using a Hashed Shard Key* (page 711).

Querying The `mongos` provides an interface for applications to interact with sharded clusters that hides the complexity of *data partitioning*. A `mongos` receives queries from applications, and uses metadata from the *config server* (page 684), to route queries to the `mongod` instances with the appropriate data. While the `mongos` succeeds in making all querying operational in sharded environments, the *shard key* you select can have a profound affect on query performance.

See also:

The *Sharded Cluster Query Routing* (page 692) and *config server* (page 684) sections for a more general overview of querying in sharded environments.

Query Isolation Generally, the fastest queries in a sharded environment are those that `mongos` will route to a single shard, using the *shard key* and the cluster meta data from the *config server* (page 684). For queries that don't include the shard key, `mongos` must query all shards, wait for their responses 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 [9], the `mongos` can route the query directly to a single shard, or a small number of shards, which provides better performance. Even if you query values of the shard key that reside in different chunks, the `mongos` will route queries directly to specific shards.

To select a shard key for a collection:

- determine the most commonly included fields in queries for a given application
- find which of these operations are most performance dependent.

If this field has low cardinality (i.e not sufficiently selective) you should add a second field to the shard key making a compound shard key. The data may become more splittable with a compound shard key.

See

Sharded Cluster Query Routing (page 692) for more information on query operations in the context of sharded clusters.

Sorting In sharded systems, the `mongos` performs a merge-sort of all sorted query results from the shards. See *Sharded Cluster Query Routing* (page 692) and *Use Indexes to Sort Query Results* (page 553) for more information.

Indivisible Chunks An insufficiently granular shard key can result in chunks that are "unsplittable". See *Create a Shard Key that is Easily Divisible* (page 710) for more information.

Additional Information

- *Considerations for Selecting Shard Keys* (page 709)
- *Shard a Collection Using a Hashed Shard Key* (page 711).

[9] In many ways, you can think of the shard key a cluster-wide index. However, be aware that sharded systems cannot enforce cluster-wide unique indexes *unless* the unique field is in the shard key. Consider the *Index Concepts* (page 485) page for more information on indexes and compound indexes.

Sharded Cluster High Availability

On this page

A *production* (page 686) *cluster* has no single point of failure. This section introduces the availability concerns for MongoDB deployments in general and highlights potential failure scenarios and available resolutions.

Application Servers or mongos Instances Become Unavailable

If each application server has its own mongos instance, other application servers can continue access the database. Furthermore, mongos instances do not maintain persistent state, and they can restart and become unavailable without losing any state or data. When a mongos instance starts, it retrieves a copy of the *config database* and can begin routing queries.

A Single mongod Becomes Unavailable in a Shard

Replica sets (page 563) provide high availability for shards. If the unavailable mongod is a *primary*, then the replica set will *elect* (page 583) a new primary. If the unavailable mongod is a *secondary*, and it disconnects the primary and secondary will continue to hold all data. In a three member replica set, even if a single member of the set experiences catastrophic failure, two other members have full copies of the data. [10]

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 Servers Become Unavailable

Three distinct mongod instances provide the *config servers* (page 684).

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 migration* (page 698) or *chunk splits* (page 738) will occur until all three servers are available. Replace the config server as soon as possible. If all config databases become unavailable, the cluster can become inoperable.

If the config servers are inconsistent, the balancer will not perform any *chunk migration* (page 698) nor will the mongos perform *auto-chunk splits* (page 738).

[10] 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*.

Note: All config servers must be running and available when you first initiate a *sharded cluster*.

Renaming Config Servers and Cluster Availability

If the name or address that a sharded cluster uses to connect to a config server changes, you must restart **every** `mongod` and `mongos` instance in the sharded cluster. Avoid downtime by using CNAMEs to identify config servers within the MongoDB deployment.

To avoid downtime when renaming config servers, use DNS names unrelated to physical or virtual hostnames to refer to your *config servers* (page 684).

Generally, refer to each config server using the DNS alias (e.g. a CNAME record). When specifying the config server connection string to `mongos`, use these names. These records make it possible to change the IP address or rename config servers without changing the connection string and without having to restart the entire cluster.

Shard Keys and Cluster Availability

The most important consideration when choosing a *shard key* are:

- to ensure that MongoDB will be able to distribute data evenly among shards, and
- to scale writes across the cluster, and
- to ensure that `mongos` can isolate most queries to a specific `mongod`.

Furthermore:

- Each shard should be a *replica set*, if a specific `mongod` instance fails, the replica set members will elect another to be *primary* and continue operation. However, if an entire shard is unreachable or fails for some reason, that data will be unavailable.
- If the shard key allows the `mongos` to isolate most operations to a single shard, then the failure of a single shard will only render *some* data unavailable.
- If your shard key distributes data required for every operation throughout the cluster, then the failure of the entire shard will render the entire cluster unavailable.

In essence, this concern for reliability simply underscores the importance of choosing a shard key that isolates query operations to a single shard.

Sharded Cluster Query Routing

On this page

MongoDB `mongos` instances route queries and write operations to *shards* in a sharded cluster. `mongos` provide the only interface to a sharded cluster from the perspective of applications. Applications never connect or communicate directly with the shards.

The mongos tracks what data is on which shard by caching the metadata from the *config servers* (page 684). The mongos uses the metadata to route operations from applications and clients to the mongod instances. A mongos has no *persistent* state and consumes minimal system resources.

The most common practice is to run mongos instances on the same systems as your application servers, but you can maintain mongos instances on the shards or on other dedicated resources.

Note: Changed in version 2.1.

Some aggregation operations using the aggregate command (i.e. db.collection.aggregate()) will cause mongos instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architecture decisions if you use the *aggregation framework* extensively in a sharded environment.

Routing Process

A mongos instance uses the following processes to route queries and return results.

How mongos Determines which Shards Receive a Query A mongos instance routes a query to a *cluster* by:

1. Determining the list of *shards* that must receive the query.
2. Establishing a cursor on all targeted shards.

In some cases, when the *shard key* or a prefix of the shard key is a part of the query, the mongos can route the query to a subset of the shards. Otherwise, the mongos must direct the query to *all* shards that hold documents for that collection.

Example

Given the following shard key:

```
{ zipcode: 1, u_id: 1, c_date: 1 }
```

Depending on the distribution of chunks in the cluster, the mongos may be able to target the query at a subset of shards, if the query contains the following fields:

```
{ zipcode: 1 }
{ zipcode: 1, u_id: 1 }
{ zipcode: 1, u_id: 1, c_date: 1 }
```

How mongos Handles Query Modifiers If the result of the query is not sorted, the mongos instance opens a result cursor that "round robins" results from all cursors on the shards.

Changed in version 2.0.5: In versions prior to 2.0.5, the mongos exhausted each cursor, one by one.

If the query specifies sorted results using the sort() cursor method, the mongos instance passes the $orderby option to the shards. The primary shard for the database receives and performs a merge sort for all results before returning the data to the client via the mongos.

If the query limits the size of the result set using the limit() cursor method, the mongos instance passes that limit to the shards and then re-applies the limit to the result before returning the result to the client.

If the query specifies a number of records to *skip* using the skip() cursor method, the mongos *cannot* pass the skip to the shards, but rather retrieves unskipped results from the shards and skips the appropriate number of documents when assembling the complete result. However, when used in conjunction with a limit(), the mongos will pass the *limit* plus the value of the skip() to the shards to improve the efficiency of these operations.

Detect Connections to mongos Instances

To detect if the MongoDB instance that your client is connected to is `mongos`, use the `isMaster` command. When a client connects to a `mongos`, `isMaster` returns a document with a `msg` field that holds the string `isdbgrid`. For example:

```
{
   "ismaster" : true,
   "msg" : "isdbgrid",
   "maxBsonObjectSize" : 16777216,
   "ok" : 1
}
```

If the application is instead connected to a `mongod`, the returned document does not include the `isdbgrid` string.

Broadcast Operations and Targeted Operations

In general, operations in a sharded environment are either:

- Broadcast to all shards in the cluster that hold documents in a collection
- Targeted at a single shard or a limited group of shards, based on the shard key

For best performance, use targeted operations whenever possible. While some operations must broadcast to all shards, you can ensure MongoDB uses targeted operations whenever possible by always including the shard key.

Broadcast Operations `mongos` instances broadcast queries to all shards for the collection **unless** the `mongos` can determine which shard or subset of shards stores this data.

Multi-update operations are always broadcast operations.

The `remove()` operation is always a broadcast operation, unless the operation specifies the shard key in full.

Targeted Operations All `insert()` operations target to one shard.

All single `update()` (including *upsert* operations) and `remove()` operations must target to one shard.

Important: All `update()` and `remove()` operations for a sharded collection that specify the `justOne` or `multi: false` option must include the *shard key or* the `_id` field in the query specification. `update()` and `remove()` operations specifying `justOne` or `multi: false` in a sharded collection without the *shard key or* the `_id` field return an error.

For queries that include the shard key or portion of the shard key, `mongos` can target the query at a specific shard or set of shards. This is the case only if the portion of the shard key included in the query is a *prefix* of the shard key. For example, if the shard key is:

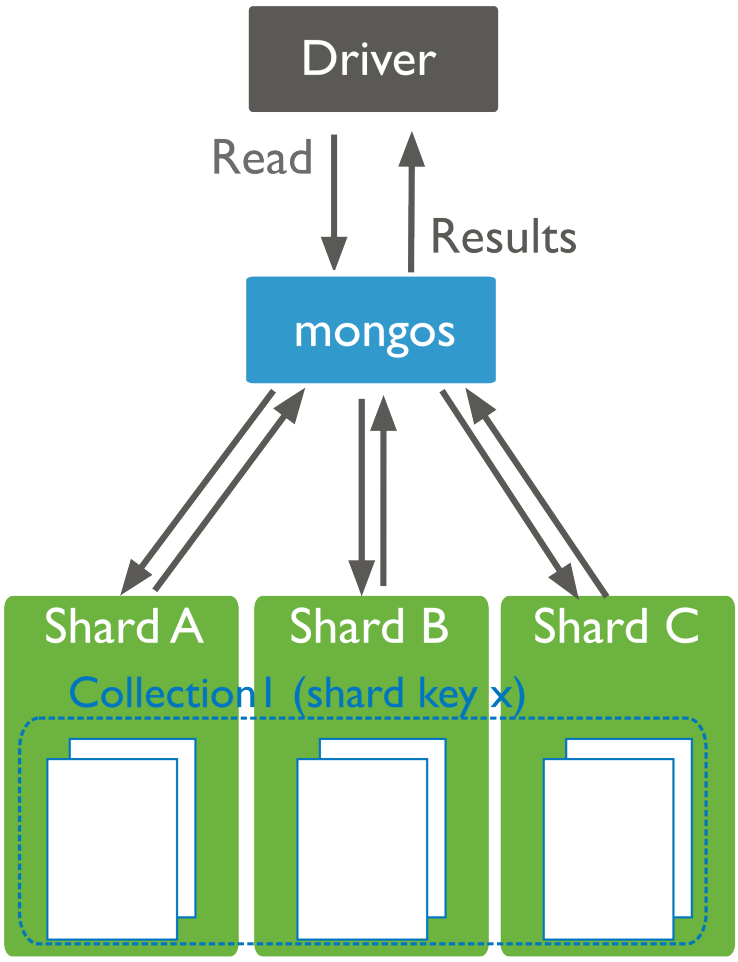
```
{ a: 1, b: 1, c: 1 }
```

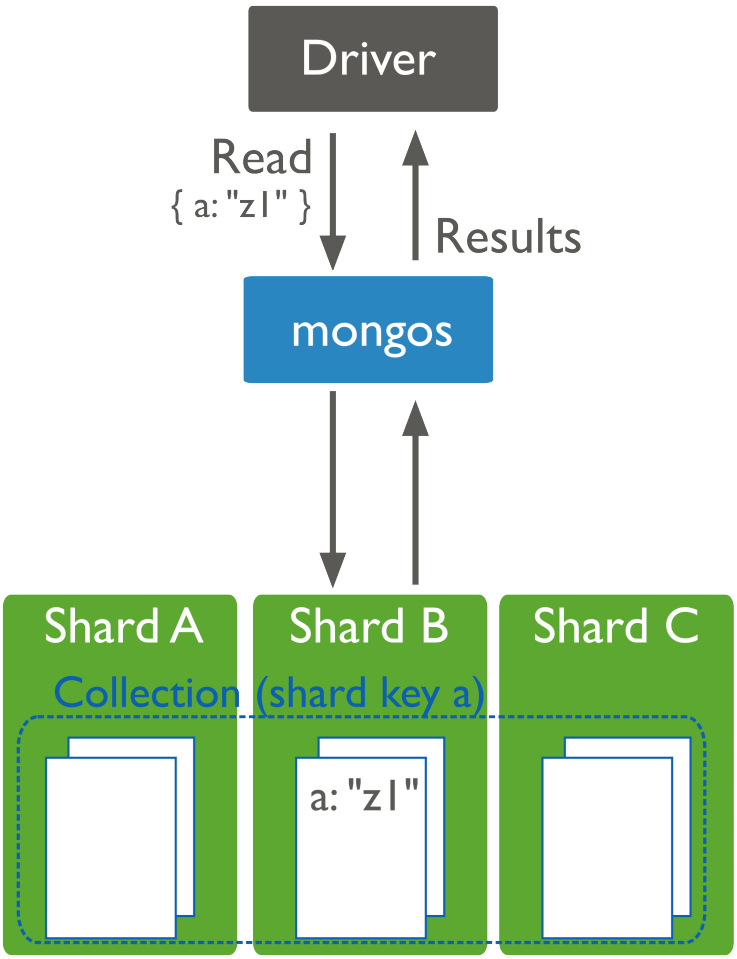
The `mongos` program *can* route queries that include the full shard key or either of the following shard key prefixes at a specific shard or set of shards:

```
{ a: 1 }
{ a: 1, b: 1 }
```

Depending on the distribution of data in the cluster and the selectivity of the query, `mongos` may still have to contact multiple shards [11] to fulfill these queries.

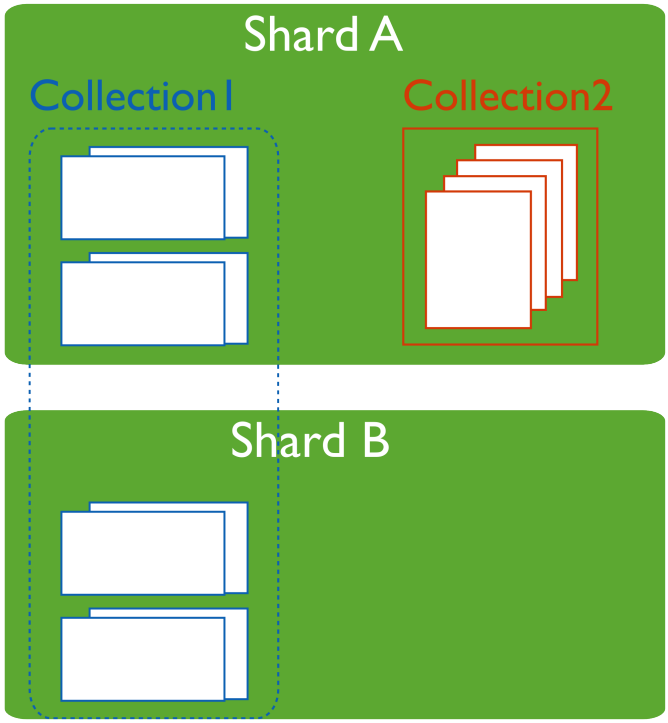
[11] `mongos` will route some queries, even some that include the shard key, to all shards, if needed.


Driver
Read
Results
mongos
Shard A
Shard B
Shard C
Collection 1 (shard key x)


Driver
Read
{ a: "z | " }
Results
mongos
Shard A
Shard B
Shard C
Collection (shard key a)
a: "z | "

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. [12] 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` even for operations that do not impact the sharded data.

10.2.4 Sharding Mechanics

The following documents describe sharded cluster processes.

Sharded Collection Balancing **(page 698)** Balancing distributes a sharded collection's data cluster to all of the shards.

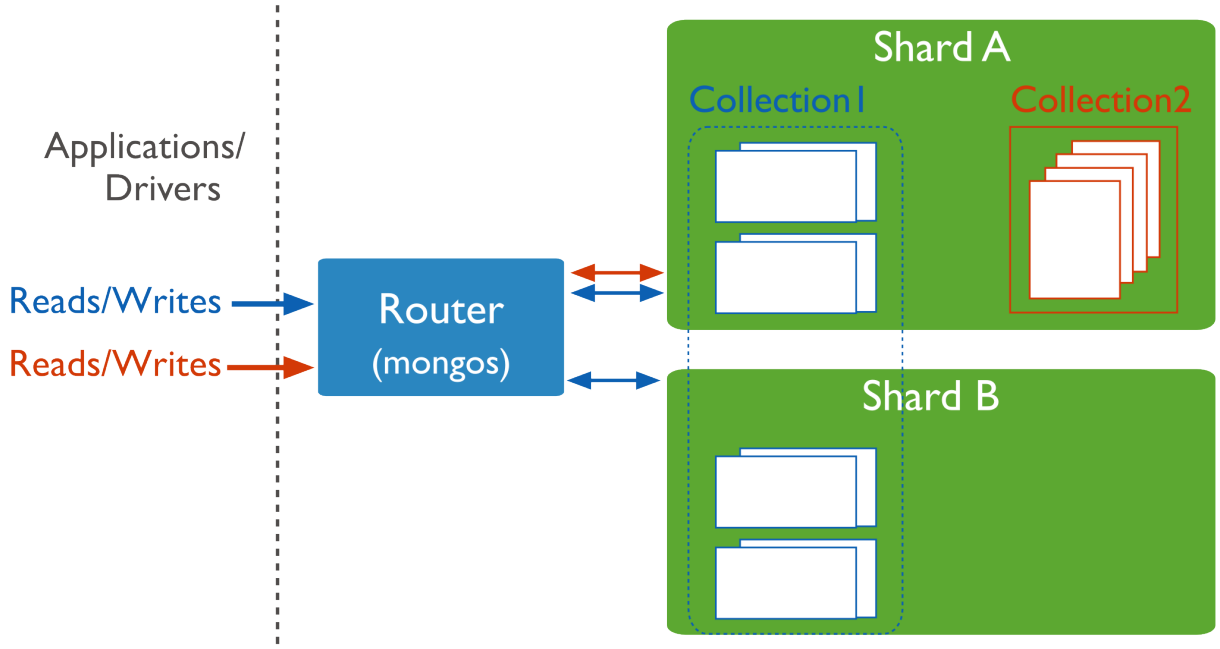
Chunk Migration Across Shards **(page 700)** MongoDB migrates chunks to shards as part of the balancing process.

Chunk Splits in a Sharded Cluster **(page 702)** When a chunk grows beyond the configured size, MongoDB splits the chunk in half.

Shard Key Indexes **(page 703)** Sharded collections must keep an index that starts with the shard key.

Sharded Cluster Metadata **(page 703)** The cluster maintains internal metadata that reflects the location of data within the cluster.

[12] As you configure sharding, you will use the `enableSharding` command to enable sharding for a database. This simply makes it possible to use the `shardCollection` command on a collection within that database.



Sharded Collection Balancing

On this page

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 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 enabled.

Any `mongos` instance in the cluster can start a balancing round. When a balancer process is active, the responsible `mongos` acquires a "lock" by modifying a document in the `lock` collection in the *Config Database* (page 754).

Note: Changed in version 2.0: Before MongoDB version 2.0, large differences in timekeeping (i.e. clock skew) between `mongos` instances could lead to failed distributed locks. This carries the possibility of data loss, particularly with skews larger than 5 minutes. Always use the network time protocol (NTP) by running `ntpd` on your servers to minimize clock skew.

To address uneven chunk distribution for a sharded collection, the balancer *migrates chunks* (page 700) 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 distribution of chunks for the collection across the shards. For details about chunk migration, see *Chunk Migration Procedure* (page 700).

Changed in version 2.6: Chunk migrations can have an impact on disk space. Starting in MongoDB 2.6, the source shard automatically archives the migrated documents by default. For details, see *moveChunk directory* (page 701).

Chunk migrations carry some overhead in terms of bandwidth and workload, both of which can impact database performance. The *balancer* attempts to minimize the impact by:

- Moving only one chunk at a time. See also *Chunk Migration Queuing* (page 701).
- 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 699).

You may disable the balancer temporarily for maintenance. See *Disable the Balancer* (page 732) 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 731) for details.

Note: The specification of the balancing window is relative to the local time zone of all individual `mongos` instances in the cluster.

See also:

Manage Sharded Cluster Balancer (page 730).

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
Fewer than 20	2
20-79	4
80 and greater	8

Once a balancing round starts, the balancer will not stop until, for the collection, the difference between the number of chunks on any two shards for that collection is *less than two* or a chunk migration fails.

Shard Size

By default, MongoDB will attempt to fill all available disk space with data on every shard as the data set grows. To ensure that the cluster always has the capacity to handle data growth, monitor disk usage as well as other performance metrics.

When adding a shard, you may set a "maximum size" for that shard. This prevents the *balancer* from migrating chunks to the shard when the value of `mapped` exceeds the "maximum size". Use the `maxSize` parameter of the `addShard` command to set the "maximum size" for the shard.

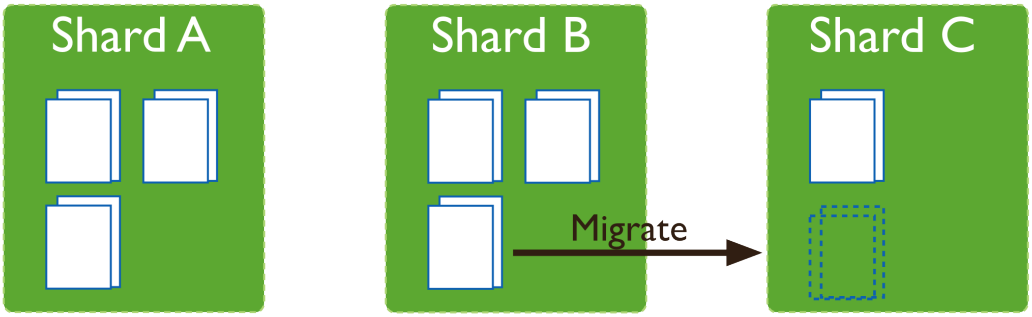
See also:

Change the Maximum Storage Size for a Given Shard (page 729) and *Monitoring for MongoDB* (page 195).

Chunk Migration Across Shards

On this page

Chunk migration moves the chunks of a sharded collection from one shard to another and is part of the *balancer* (page 698) process.



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 739) for more details.
- Automatic. The *balancer* (page 698) process automatically migrates chunks when there is an uneven distribution of a sharded collection's chunks across the shards. See *Migration Thresholds* (page 699) for more details.

Chunk Migration Procedure All chunk migrations use the following procedure:

1. The balancer process sends the `moveChunk` command to the source shard.
2. The source starts the move with an internal `moveChunk` command. During the migration process, operations to the chunk route to the source shard. The source shard is responsible for incoming write operations for the chunk.
3. The destination shard builds any indexes required by the source that do not exist on the destination.
4. The destination shard begins requesting documents in the chunk and starts receiving copies of the data.
5. 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.
6. When fully synchronized, the destination shard connects to the *config database* and updates the cluster metadata with the new location for the chunk.
7. After the destination shard completes the update of the metadata, and once there are no open cursors on the chunk, the source shard deletes its copy of the documents.

Changed in version 2.6: The source shard automatically archives the migrated documents by default. For more information, see *moveChunk directory* (page 701).

Changed in version 2.4: If the balancer needs to perform additional chunk migrations from the source shard, the balancer can start the next chunk migration without waiting for the current migration process to finish this deletion step. See *Chunk Migration Queuing* (page 701).

The migration process ensures consistency and maximizes the availability of chunks during balancing.

Chunk Migration Queuing Changed in version 2.4.

To migrate multiple chunks from a shard, the balancer migrates the chunks one at a time. However, the balancer does not wait for the current migration's delete phase to complete before starting the next chunk migration. See *Chunk Migration* (page 700) for the chunk migration process and the delete phase.

This queuing behavior allows shards to unload chunks more quickly in cases of heavily imbalanced cluster, such as when performing initial data loads without pre-splitting and when adding new shards.

This behavior also affect the `moveChunk` command, and migration scripts that use the `moveChunk` command may proceed more quickly.

In some cases, the delete phases may persist longer. If multiple delete phases are queued but not yet complete, a crash of the replica set's primary can orphan data from multiple migrations.

Chunk Migration and Replication By default, each document operation during chunk migration propagates to at least one secondary before the balancer proceeds with the next document.

To override this behavior and allow the balancer to continue without waiting for replication to a secondary, set the `_secondaryThrottle` parameter to `false`. See *Change Replication Behavior for Chunk Migration (Secondary Throttle)* (page 730) to update the `_secondaryThrottle` parameter for the balancer.

When called directly, `moveChunk` does *not* require that operations propagate to shards during operation: its `secondaryThrottle` defaults to `false`.

Independent of the `secondaryThrottle` setting, certain phases of the chunk migration have the following replication policy:

- MongoDB briefly pauses all application writes to the source shard before updating the config servers with the new location for the chunk, and resumes the application writes after the update. The chunk move requires all writes to be acknowledged by majority of the members of the replica set both before and after committing the chunk move to config servers.
- When an outgoing chunk migration finishes and cleanup occurs, all writes must be replicated to a majority of servers before further cleanup (from other outgoing migrations) or new incoming migrations can proceed.

Changed in version 2.4: In previous versions, the balancer did not wait for the document move to replicate to a secondary. For details, see Secondary Throttle in the v2.2 Manual[13]

`moveChunk` directory

Starting in MongoDB 2.6, `sharding.archiveMovedChunks` is enabled by default. With `sharding.archiveMovedChunks` enabled, the source shard archives the documents in the migrated chunks in a directory named after the collection namespace under the `moveChunk` directory in the `storage.dbPath`.

[13] http://docs.mongodb.org/v2.2/tutorial/configure-sharded-cluster-balancer/#sharded-cluster-config-secondary-throttle

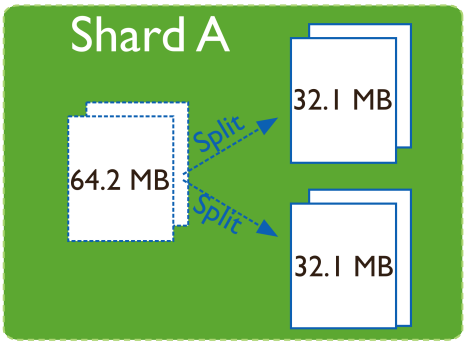
Jumbo Chunks

During chunk migration, if the chunk exceeds the *specified chunk size* (page 702) or if the number of documents in the chunk exceeds `Maximum Number of Documents Per Chunk to Migrate`, MongoDB does not migrate the chunk. Instead, MongoDB attempts to *split* (page 702) the chunk. If the split is unsuccessful, MongoDB labels the chunk as *jumbo* to avoid repeated attempts to migrate the chunk.

Chunk Splits in a Sharded Cluster

On this page

As chunks grow beyond the *specified chunk size* (page 702) a `mongos` instance will attempt to split the chunk in half. Splits may lead to an uneven distribution of the chunks for a collection across the shards. In such cases, the `mongos` instances will initiate a round of migrations to redistribute chunks across shards. See *Sharded Collection Balancing* (page 698) for more details on balancing chunks across shards.



Chunk Size

The default *chunk* size in MongoDB is 64 megabytes. You can *increase or reduce the chunk size* (page 743), mindful of its effect on the cluster's efficiency.

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations. This creates expense at the query routing (`mongos`) layer.
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.
3. Chunk size affects the `Maximum Number of Documents Per Chunk to Migrate`.

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.

Indivisible Chunks

In some cases, chunks can grow beyond the *specified chunk size* (page 702) but cannot undergo a split; e.g. if a chunk represents a single shard key value. See *Considerations for Selecting Shard Keys* (page 709) for considerations for selecting a shard key.

Shard Key Indexes

All sharded collections **must** have an index that starts with the *shard key*. If you shard a collection without any documents and *without* such an index, the `shardCollection` command will create the index on the shard key. If the collection already has documents, you must create the index before using `shardCollection`.

Changed in version 2.2: The index on the shard key no longer needs to be only on the shard key. This index can be an index of the shard key itself, or a *compound index* where the shard key is a prefix of the index.

Important: The index on the shard key **cannot** be a *multikey index* (page 491).

A sharded collection named `people` has for its shard key the field `zipcode`. It currently has the index `{ zipcode: 1 }`. You can replace this index with a compound index `{ zipcode: 1, username: 1 }`, as follows:

1. Create an index on `{ zipcode: 1, username: 1 }`:

   ```
   db.people.ensureIndex( { zipcode: 1, username: 1 } );
   ```

2. When MongoDB finishes building the index, you can safely drop the existing index on `{ zipcode: 1 }`:

   ```
   db.people.dropIndex( { zipcode: 1 } );
   ```

Since the index on the shard key cannot be a multikey index, the index `{ zipcode: 1, username: 1 }` can only replace the index `{ zipcode: 1 }` if there are no array values for the `username` field.

If you drop the last valid index for the shard key, recover by recreating an index on just the shard key.

For restrictions on shard key indexes, see *limits-shard-keys*.

Sharded Cluster Metadata

Config servers (page 684) store the metadata for a sharded cluster. The metadata reflects state and organization of the sharded data sets and system. The metadata includes the list of chunks on every shard and the ranges that define the chunks. The `mongos` instances cache this data and use it to route read and write operations to shards.

Config servers store the metadata in the *Config Database* (page 754).

Important: Always back up the `config` database before doing any maintenance on the config server.

To access the `config` database, issue the following command from the `mongo` shell:

```
use config
```

In general, you should *never* edit the content of the `config` database directly. The `config` database contains the following collections:

- `changelog` (page 755)
- `chunks` (page 756)
- `collections` (page 757)
- `databases` (page 757)
- `lockpings` (page 757)
- `locks` (page 758)
- `mongos` (page 758)
- `settings` (page 758)
- `shards` (page 759)
- `version` (page 759)

For more information on these collections and their role in sharded clusters, see *Config Database* (page 754). See *Read and Write Operations on Config Servers* (page 684) for more information about reads and updates to the metadata.

10.3 Sharded Cluster Tutorials

The following tutorials provide instructions for administering *sharded clusters*. For a higher-level overview, see *Sharding* (page 675).

Sharded Cluster Deployment Tutorials **(page 705)** Instructions for deploying sharded clusters, adding shards, selecting shard keys, and the initial configuration of sharded clusters.

> ***Deploy a Sharded Cluster*** **(page 705)** 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 709)** 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 711)** Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.
>
> ***Add Shards to a Cluster*** **(page 712)** Add a shard to add capacity to a sharded cluster.
>
> Continue reading from *Sharded Cluster Deployment Tutorials* (page 705) for additional tutorials.

Sharded Cluster Maintenance Tutorials **(page 720)** Procedures and tasks for common operations on active sharded clusters.

> ***View Cluster Configuration*** **(page 721)** View status information about the cluster's databases, shards, and chunks.
>
> ***Remove Shards from an Existing Sharded Cluster*** **(page 734)** Migrate a single shard's data and remove the shard.

Migrate Config Servers with Different Hostnames **(page 723)** Migrate a config server to a new system that uses a new hostname. If possible, avoid changing the hostname and instead use the *Migrate Config Servers with the Same Hostname* (page 722) procedure.

Manage Shard Tags **(page 747)** Use tags to associate specific ranges of shard key values with specific shards.

Continue reading from *Sharded Cluster Maintenance Tutorials* (page 720) for additional tutorials.

Sharded Cluster Data Management **(page 737)** Practices that address common issues in managing large sharded data sets.

Troubleshoot Sharded Clusters **(page 752)** Presents solutions to common issues and concerns relevant to the administration and use of sharded clusters. Refer to *FAQ: MongoDB Diagnostics* (page 799) for general diagnostic information.

10.3.1 Sharded Cluster Deployment Tutorials

The following tutorials provide information on deploying sharded clusters.

Deploy a Sharded Cluster **(page 705)** 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 709)** 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 711)** Shard a collection based on hashes of a field's values in order to ensure even distribution over the collection's shards.

Add Shards to a Cluster **(page 712)** Add a shard to add capacity to a sharded cluster.

Deploy Three Config Servers for Production Deployments **(page 713)** Convert a test deployment with one config server to a production deployment with three config servers.

Convert a Replica Set to a Replicated Sharded Cluster **(page 714)** Convert a replica set to a sharded cluster in which each shard is its own replica set.

Convert Sharded Cluster to Replica Set **(page 719)** Replace your sharded cluster with a single replica set.

See also:

Enable Authentication in a Sharded Cluster (page 354)

Deploy a Sharded Cluster

On this page

Use the following sequence of tasks to deploy a sharded cluster:

Warning: Sharding and "localhost" Addresses
If you use either "localhost" or `127.0.0.1` as the hostname portion of any host identifier, for example as the `host` argument to `addShard` or the value to the `--configdb` run time option, then you must use "localhost" or `127.0.0.1` for *all* host settings for any MongoDB instances in the cluster. If you mix localhost addresses and remote host address, MongoDB will error.

Start the Config Server Database Instances

The config server processes are `mongod` instances that store the cluster's metadata. You designate a `mongod` as a config server using the `--configsvr` option. Each config server stores a complete copy of the cluster's metadata.

In production deployments, you must deploy exactly three config server instances, each running on different servers to assure good uptime and data safety. In test environments, you can run all three instances on a single server.

Important: All members of a sharded cluster must be able to connect to *all* other members of a sharded cluster, including all shards and all config servers. Ensure that the network and security systems including all interfaces and firewalls, allow these connections.

1. Create data directories for each of the three config server instances. By default, a config server stores its data files in the */data/configdb* directory. You can choose a different location. To create a data directory, issue a command similar to the following:

   ```
   mkdir /data/configdb
   ```

2. Start the three config server instances. Start each by issuing a command using the following syntax:

   ```
   mongod --configsvr --dbpath <path> --port <port>
   ```

   The default port for config servers is `27019`. You can specify a different port. The following example starts a config server using the default port and default data directory:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27019
   ```

   For additional command options, see `http://docs.mongodb.org/manual/reference/program/mongod` or `http://docs.mongodb.org/manual/reference/configuration-options`.

   Note: All config servers must be running and available when you first initiate a *sharded cluster*.

Start the `mongos` Instances

The `mongos` instances are lightweight and do not require data directories. You can run a `mongos` instance on a system that runs other cluster components, such as on an application server or a server running a `mongod` process. By default, a `mongos` instance runs on port `27017`.

When you start the `mongos` instance, specify the hostnames of the three config servers, either in the configuration file or as command line parameters.

Tip

To avoid downtime, give each config server a logical DNS name (unrelated to the server's physical or virtual hostname). Without logical DNS names, moving or renaming a config server requires shutting down every `mongod` and `mongos` instance in the sharded cluster.

To start a mongos instance, issue a command using the following syntax:

```
mongos --configdb <config server hostnames>
```

For example, to start a mongos that connects to config server instance running on the following hosts and on the default ports:

- cfg0.example.net
- cfg1.example.net
- cfg2.example.net

You would issue the following command:

```
mongos --configdb cfg0.example.net:27019,cfg1.example.net:27019,cfg2.example.net:27019
```

Each mongos in a sharded cluster must use the same configDB string, with identical host names listed in identical order.

If you start a mongos instance with a string that *does not* exactly match the string used by the other mongos instances in the cluster, the mongos return a *Config Database String Error* (page 752) error and refuse to start.

Add Shards to the Cluster

A *shard* can be a standalone mongod or a *replica set*. In a production environment, each shard should be a replica set. Use the procedure in *Deploy a Replica Set* (page 607) to deploy replica sets for each shard.

1. From a mongo shell, connect to the mongos instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

   For example, if a mongos is accessible at mongos0.example.net on port 27017, issue the following command:

   ```
   mongo --host mongos0.example.net --port 27017
   ```

2. Add each shard to the cluster using the sh.addShard() method, as shown in the examples below. Issue sh.addShard() separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the addShard database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see addShard.

   The following are examples of adding a shard with sh.addShard():

   - To add a shard for a replica set named rs1 with a member running on port 27017 on mongodb0.example.net, issue the following command:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017" )
     ```

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
     ```

- To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

```
sh.addShard( "mongodb0.example.net:27017" )
```

Note: It might take some time for *chunks* to migrate to the new shard.

Enable Sharding for a Database

Before you can shard a collection, you must enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but make it possible to shard the collections in that database.

Once you enable sharding for a database, MongoDB assigns a *primary shard* for that database where MongoDB stores all data before sharding begins.

1. From a `mongo` shell, connect to the `mongos` instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

2. Issue the `sh.enableSharding()` method, specifying the name of the database for which to enable sharding. Use the following syntax:

   ```
   sh.enableSharding("<database>")
   ```

Optionally, you can enable sharding for a database using the `enableSharding` command, which uses the following syntax:

```
db.runCommand( { enableSharding: <database> } )
```

Enable Sharding for a Collection

You enable sharding on a per-collection basis.

1. Determine what you will use for the *shard key*. Your selection of the shard key affects the efficiency of sharding. See the selection considerations listed in the *Considerations for Selecting Shard Key* (page 710).
2. If the collection already contains data you must create an index on the *shard key* using `ensureIndex()`. If the collection is empty then MongoDB will create the index as part of the `sh.shardCollection()` step.
3. Enable sharding for a collection by issuing the `sh.shardCollection()` method in the `mongo` shell. The method uses the following syntax:

   ```
   sh.shardCollection("<database>.<collection>", shard-key-pattern)
   ```

   Replace the `<database>.<collection>` string with the full namespace of your database, which consists of the name of your database, a dot (e.g. `.`), and the full name of the collection. The `shard-key-pattern` represents your shard key, which you specify in the same form as you would an `index` key pattern.

   Example

   The following sequence of commands shards four collections:

   ```
   sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
   sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
   sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
   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 710) 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 710) 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 710) 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 524), which should provide an even distribution of documents across a cluster.

Considerations for Selecting Shard Keys

Choosing a Shard Key

For many collections there may be no single, naturally occurring key that possesses all the qualities of a good shard key. The following strategies may help construct a useful shard key from existing data:

1. Compute a more ideal shard key in your application layer, and store this in all of your documents, potentially in the `_id` field.
2. Use a compound shard key that uses two or three values from all documents that provide the right mix of cardinality with scalable write operations and query isolation.
3. Determine that the impact of using a less than ideal shard key is insignificant in your use case, given:
   - limited write volume,
   - expected data size, or
   - application query patterns.
4. New in version 2.4: Use a *hashed shard key*. Choose a field that has high cardinality and create a *hashed index* (page 524) on that field. MongoDB uses these hashed index values as shard key values, which ensures an even distribution of documents across the shards.

Tip

MongoDB automatically computes the hashes when resolving queries using hashed indexes. Applications do **not** need to compute hashes.

Considerations for Selecting Shard Key

Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster. Appropriate shard key choice depends on the schema of your data and the way that your applications query and write data.

Create a Shard Key that is Easily Divisible An easily divisible shard key makes it easy for MongoDB to distribute content among the shards. Shard keys that have a limited number of possible values can result in chunks that are "unsplittable".

For instance, if a chunk represents a single shard key value, then MongoDB cannot split the chunk even when the chunk exceeds the size at which *splits* (page 702) occur.

See also:

Cardinality (page 710)

Create a Shard Key that has High Degree of Randomness A shard key with high degree of randomness prevents any single shard from becoming a bottleneck and will distribute write operations among the cluster.

See also:

Write Scaling (page 689)

Create a Shard Key that Targets a Single Shard A shard key that targets a single shard makes it possible for the **mongos** program to return most query operations directly from a single *specific* **mongod** instance. Your shard key should be the primary field used by your queries. Fields with a high degree of "randomness" make it difficult to target operations to specific shards.

See also:

Query Isolation (page 690)

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 690) or appropriate *write scaling* (page 689).

If you choose a shard key with low cardinality, some chunks may grow too large for MongoDB to migrate. See *Jumbo Chunks* (page 702) for more information.

Shard Key Selection Strategy

When selecting a shard key, it is difficult to balance the qualities of an ideal shard key, which sometimes dictate opposing strategies. For instance, it's difficult to produce a key that has both a high degree randomness for even data distribution and a shard key that allows your application to target specific shards. For some workloads, it's more important to have an even data distribution, and for others targeted queries are essential.

Therefore, the selection of a shard key is about balancing both your data and the performance characteristics caused by different possible data distributions and system workloads.

Shard a Collection Using a Hashed Shard Key

On this page

New in version 2.4.

Hashed shard keys (page 689) use a *hashed index* (page 524) of a field as the *shard key* to partition data across your sharded cluster.

For suggestions on choosing the right field as your hashed shard key, see *Hashed Shard Keys* (page 689). For limitations on hashed indexes, see *Create a Hashed Index* (page 524).

Note: If chunk migrations are in progress while creating a hashed shard key collection, the initial chunk distribution may be uneven until the balancer automatically balances the collection.

Shard the Collection

To shard a collection using a hashed shard key, use an operation in the `mongo` that resembles the following:

```
sh.shardCollection( "records.active", { a: "hashed" } )
```

This operation shards the `active` collection in the `records` database, using a hash of the `a` field as the shard key.

Specify the Initial Number of Chunks

If you shard an empty collection using a hashed shard key, MongoDB automatically creates and migrates empty chunks so that each shard has two chunks. To control how many chunks MongoDB creates when sharding the collection, use `shardCollection` with the `numInitialChunks` parameter.

Important: MongoDB 2.4 adds support for hashed shard keys. After sharding a collection with a hashed shard key, you must use the MongoDB 2.4 or higher `mongos` and `mongod` instances in your sharded cluster.

Warning: MongoDB `hashed` indexes truncate floating point numbers to 64-bit integers before hashing. For example, a `hashed` index would store the same value for a field that held a value of `2.3`, `2.2`, and `2.9`. To prevent collisions, do not use a `hashed` index for floating point numbers that cannot be reliably converted to 64-bit integers (and then back to floating point). MongoDB `hashed` indexes do not support floating point values larger than 2^{53}.

Add Shards to a Cluster

On this page

You add shards to a *sharded cluster* after you create the cluster or any time that you need to add capacity to the cluster. If you have not created a sharded cluster, see *Deploy a Sharded Cluster* (page 705).

In production environments, all shards should be *replica sets*.

Considerations

Balancing When you add a shard to a sharded cluster, you affect the balance of *chunks* among the shards of a cluster for all existing sharded collections. The balancer will begin migrating chunks so that the cluster will achieve balance. See *Sharded Collection Balancing* (page 698) for more information.

Changed in version 2.6: Chunk migrations can have an impact on disk space. Starting in MongoDB 2.6, the source shard automatically archives the migrated documents by default. For details, see *moveChunk directory* (page 701).

Capacity Planning When adding a shard to a cluster, always ensure that the cluster has enough capacity to support the migration required for balancing the cluster without affecting legitimate production traffic.

Add a Shard to a Cluster

You interact with a sharded cluster by connecting to a `mongos` instance.

1. From a `mongo` shell, connect to the `mongos` instance. For example, if a `mongos` is accessible at `mongos0.example.net` on port `27017`, issue the following command:

```
mongo --host mongos0.example.net --port 27017
```

2. Add a shard to the cluster using the `sh.addShard()` method, as shown in the examples below. Issue `sh.addShard()` separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the `addShard` database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see `addShard`.

   The following are examples of adding a shard with `sh.addShard()`:

   - To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017" )
     ```

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
     ```

   - To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "mongodb0.example.net:27017" )
     ```

   Note: It might take some time for *chunks* to migrate to the new shard.

Deploy Three Config Servers for Production Deployments

This procedure converts a test deployment with only one *config server* (page 684) to a production deployment with three config servers.

Tip

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

For redundancy, all production *sharded clusters* (page 675) should deploy three config servers on three different machines. Use a single config server only for testing deployments, never for production deployments. When you shift to production, upgrade immediately to three config servers.

To convert a test deployment with one config server to a production deployment with three config servers:

1. Shut down all existing MongoDB processes in the cluster. This includes:
   - all `mongod` instances or *replica sets* that provide your shards.
   - all `mongos` instances in your cluster.
2. Copy the entire `dbPath` file system tree from the existing config server to the two machines that will provide the additional config servers. These commands, issued on the system with the existing *Config Database* (page 754), `mongo-config0.example.net` may resemble the following:

```
rsync -az /data/configdb mongo-config1.example.net:/data/configdb
rsync -az /data/configdb mongo-config2.example.net:/data/configdb
```

3. Start all three config servers, using the same invocation that you used for the single config server.

   ```
   mongod --configsvr
   ```

4. Restart all shard `mongod` and `mongos` processes.

Convert a Replica Set to a Replicated Sharded Cluster

On this page

Overview

This tutorial converts a single three-member replica set to a sharded cluster with two shards. Each shard is an independent three-member replica set. The procedure is as follows:

1. Create the initial three-member replica set and insert data into a collection. See *Set Up Initial Replica Set* (page 715).
2. Start the config databases and a `mongos`. See *Deploy Config Databases and mongos* (page 715).
3. Add the initial replica set as a shard. See *Add Initial Replica Set as a Shard* (page 716).
4. Create a second shard and add to the cluster. See *Add Second Shard* (page 716).
5. Shard the desired collection. See *Shard a Collection* (page 717).

Prerequisites

This tutorial uses a total of ten servers: one server for the `mongos` and three servers each for the first *replica set*, the second replica set, and the *config servers* (page 684).

Each server must have a resolvable domain, hostname, or IP address within your system.

The tutorial uses the default data directories (e.g. `/data/db` and `/data/configdb`). Create the appropriate directories with appropriate permissions. To use different paths, see `http://docs.mongodb.org/manual/reference/configuration-options`.

The tutorial uses the *default ports* (page 424) (e.g. `27017` and `27019`). To use different ports, see `http://docs.mongodb.org/manual/reference/configuration-options`.

Considerations

In production deployments, use exactly **three** config servers. Each config server must be on a separate machine.

In development and testing environments, you can deploy a cluster with a single config server.

Procedures

Set Up Initial Replica Set This procedure creates the initial three-member replica set `rs0`. The replica set members are on the following hosts: `mongodb0.example.net`, `mongodb1.example.net`, and `mongodb2.example.net`.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod`, specifying the replica set name through the `replSet` option. Include any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

```
mongod --replSet "rs0"
```

Repeat this step for the other two members of the `rs0` replica set.

Step 2: Connect a `mongo` shell to a replica set member. Connect a `mongo` shell to *one* member of the replica set (e.g. `mongodb0.example.net`)

```
mongo mongodb0.example.net
```

Step 3: Initiate the replica set. From the `mongo` shell, run `rs.initiate()` to initiate a replica set that consists of the current member.

```
rs.initiate()
```

Step 4: Add the remaining members to the replica set.

```
rs.add("mongodb1.example.net")
rs.add("mongodb2.example.net")
```

Step 5: Create and populate a new collection. The following step adds one million documents to the collection `test_collection` and can take several minutes depending on your system.

Issue the following operations on the primary of the replica set:

```
use test
var bulk = db.test_collection.initializeUnorderedBulkOp();
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina", "
for(var i=0; i<1000000; i++){
   user_id = i;
   name = people[Math.floor(Math.random()*people.length)];
   number = Math.floor(Math.random()*10001);
   bulk.insert( { "user_id":user_id, "name":name, "number":number });
}
bulk.execute();
```

For more information on deploying a replica set, see *Deploy a Replica Set* (page 607).

Deploy Config Databases and `mongos` This procedure deploys the three config servers and the `mongos`. The config servers use the following hosts: `mongodb7.example.net`, `mongodb8.example.net`, and `mongodb9.example.net`; the `mongos` uses `mongodb6.example.net`.

Step 1: Start three config databases. On each `mongodb7.example.net`, `mongodb8.example.net`, and `mongodb9.example.net` server, start the config server using default data directory `/data/configdb` and the default port `27019`:

```
mongod --configsvr
```

To modify the default settings or to include additional options specific to your deployment, see `http://docs.mongodb.org/manual/reference/configuration-options`.

Step 2: Start a `mongos` instance. On `mongodb6.example.net`, start the `mongos` specifying the config servers. The `mongos` runs on the default port `27017`.

This tutorial specifies a small *`--chunkSize`* of 1 MB to test sharding with the `test_collection` created earlier.

Note: In production environments, do **not** use a small chunkSize size.

```
mongos --configdb mongodb07.example.net:27019,mongodb08.example.net:27019,mongodb09.example.net:2701
```

Add Initial Replica Set as a Shard The following procedure adds the initial replica set `rs0` as a shard.

Step 1: Connect a `mongo` shell to the `mongos`.

```
mongo mongodb6.example.net:27017/admin
```

Step 2: Add the shard. Add a shard to the cluster with the `sh.addShard` method:

```
sh.addShard( "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net:27017" )
```

Add Second Shard The following procedure deploys a new replica set `rs1` for the second shard and adds it to the cluster. The replica set members are on the following hosts: `mongodb3.example.net`, `mongodb4.example.net`, and `mongodb5.example.net`.

Step 1: Start each member of the replica set with the appropriate options. For each member, start a `mongod`, specifying the replica set name through the `replSet` option. Include any other parameters specific to your deployment. For replication-specific parameters, see *cli-mongod-replica-set*.

```
mongod --replSet "rs1"
```

Repeat this step for the other two members of the `rs1` replica set.

Step 2: Connect a `mongo` shell to a replica set member. Connect a `mongo` shell to *one* member of the replica set (e.g. `mongodb3.example.net`)

```
mongo mongodb3.example.net
```

Step 3: Initiate the replica set. From the `mongo` shell, run `rs.initiate()` to initiate a replica set that consists of the current member.

```
rs.initiate()
```

Step 4: Add the remaining members to the replica set. Add the remaining members with the `rs.add()` method.

```
rs.add("mongodb4.example.net")
rs.add("mongodb5.example.net")
```

Step 5: Connect a `mongo` shell to the `mongos`.

```
mongo mongodb6.example.net:27017/admin
```

Step 6: Add the shard. In a `mongo` shell connected to the `mongos`, add the shard to the cluster with the `sh.addShard()` method:

```
sh.addShard( "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.example.net:27017" )
```

Shard a Collection

Step 1: Connect a `mongo` shell to the `mongos`.

```
mongo mongodb6.example.net:27017/admin
```

Step 2: Enable sharding for a database. Before you can shard a collection, you must first enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but makes it possible to shard the collections in that database.

The following operation enables sharding on the `test` database:

```
sh.enableSharding( "test" )
```

The operation returns the status of the operation:

```
{ "ok" : 1 }
```

Step 3: Determine the shard key. For the collection to shard, determine the shard key. The *shard key* (page 687) determines how MongoDB distributes the documents between shards. Good shard keys:

- have values that are evenly distributed among all documents,
- group documents that are often accessed at the same time into contiguous chunks, and
- allow for effective distribution of activity among shards.

Once you shard a collection with the specified shard key, you **cannot** change the shard key. For more information on shard keys, see *Shard Keys* (page 687) and *Considerations for Selecting Shard Keys* (page 709).

This procedure will use the `number` field as the shard key for `test_collection`.

Step 4: Create an index on the shard key. Before sharding a non-empty collection, create an *index on the shard key* (page 703).

```
use test
db.test_collection.ensureIndex( { number : 1 } )
```

Step 5: Shard the collection. In the `test` database, shard the `test_collection`, specifying `number` as the shard key.

```
use test
sh.shardCollection( "test.test_collection", { "number" : 1 } )
```

The method returns the status of the operation:

```
{ "collectionsharded" : "test.test_collection", "ok" : 1 }
```

The *balancer* (page 698) will redistribute chunks of documents when it next runs. As clients insert additional documents into this collection, the `mongos` will route the documents between the shards.

Step 6: Confirm the shard is balancing. To confirm balancing activity, run `db.stats()` or `db.printShardingStatus()` in the `test` database.

```
use test
db.stats()
db.printShardingStatus()
```

Example output of the `db.stats()`:

```
{
  "raw" : {
      "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net:27017" : {
         "db" : "test",
         "collections" : 3,
         "objects" : 989316,
         "avgObjSize" : 111.99974123535857,
         "dataSize" : 110803136,
         "storageSize" : 174751744,
         "numExtents" : 14,
         "indexes" : 2,
         "indexSize" : 57370992,
         "fileSize" : 469762048,
         "nsSizeMB" : 16,
         "dataFileVersion" : {
            "major" : 4,
            "minor" : 5
         },
         "extentFreeList" : {
            "num" : 0,
            "totalSize" : 0
         },
         "ok" : 1
      },
      "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.example.net:27017" : {
         "db" : "test",
         "collections" : 3,
         "objects" : 14697,
         "avgObjSize" : 111.98258147921345,
         "dataSize" : 1645808,
         "storageSize" : 2809856,
         "numExtents" : 7,
         "indexes" : 2,
         "indexSize" : 1169168,
         "fileSize" : 67108864,
         "nsSizeMB" : 16,
         "dataFileVersion" : {
```

```
            "major" : 4,
            "minor" : 5
         },
         "extentFreeList" : {
            "num" : 0,
            "totalSize" : 0
         },
         "ok" : 1
      }
   },
   "objects" : 1004013,
   "avgObjSize" : 111,
   "dataSize" : 112448944,
   "storageSize" : 177561600,
   "numExtents" : 21,
   "indexes" : 4,
   "indexSize" : 58540160,
   "fileSize" : 536870912,
   "extentFreeList" : {
      "num" : 0,
      "totalSize" : 0
   },
   "ok" : 1
}
```

Example output of the `db.printShardingStatus()`:

```
--- Sharding Status ---
sharding version: {
   "_id" : 1,
   "version" : 4,
   "minCompatibleVersion" : 4,
   "currentVersion" : 5,
   "clusterId" : ObjectId("5446970c04ad5132c271597c")
}
shards:
   {  "_id" : "rs0",  "host" : "rs0/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.ex
   {  "_id" : "rs1",  "host" : "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.ex
databases:
   {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
   {  "_id" : "test",  "partitioned" : true,  "primary" : "rs0" }

test.test_collection
      shard key: { "number" : 1 }
      chunks:
         rs1    5
         rs0    186
      too many chunks to print, use verbose if you want to force print
```

Run these commands for a second time to demonstrate that *chunks* are migrating from `rs0` to `rs1`.

Convert Sharded Cluster to Replica Set

On this page

This tutorial describes the process for converting a *sharded cluster* to a non-sharded *replica set*. To convert a replica set into a sharded cluster *Convert a Replica Set to a Replicated Sharded Cluster* (page 714). See the *Sharding* (page 675) documentation for more information on sharded clusters.

Convert a Cluster with a Single Shard into a Replica Set

In the case of a *sharded cluster* with only one shard, that shard contains the full data set. Use the following procedure to convert that cluster into a non-sharded *replica set*:

1. Reconfigure the application to connect to the primary member of the replica set hosting the single shard that system will be the new replica set.
2. Optionally remove the `--shardsrv` option, if your `mongod` started with this option.

   Tip

   Changing the `--shardsrv` option will change the port that `mongod` listens for incoming connections on.

The single-shard cluster is now a non-sharded *replica set* that will accept read and write operations on the data set.

You may now decommission the remaining sharding infrastructure.

Convert a Sharded Cluster into a Replica Set

Use the following procedure to transition from a *sharded cluster* with more than one shard to an entirely new *replica set*.

1. With the *sharded cluster* running, *deploy a new replica set* (page 607) in addition to your sharded cluster. The replica set must have sufficient capacity to hold all of the data files from all of the current shards combined. Do not configure the application to connect to the new replica set until the data transfer is complete.
2. Stop all writes to the *sharded cluster*. You may reconfigure your application or stop all `mongos` instances. If you stop all `mongos` instances, the applications will not be able to read from the database. If you stop all `mongos` instances, start a temporary `mongos` instance on that applications cannot access for the data migration procedure.
3. Use *mongodump and mongorestore* (page 261) to migrate the data from the `mongos` instance to the new *replica set*.

   Note: Not all collections on all databases are necessarily sharded. Do not solely migrate the sharded collections. Ensure that all databases and all collections migrate correctly.

4. Reconfigure the application to use the non-sharded *replica set* instead of the `mongos` instance.

The application will now use the un-sharded *replica set* for reads and writes. You may now decommission the remaining unused sharded cluster infrastructure.

10.3.2 Sharded Cluster Maintenance Tutorials

The following tutorials provide information in maintaining sharded clusters.

View Cluster Configuration **(page 721)** View status information about the cluster's databases, shards, and chunks.

Migrate Config Servers with the Same Hostname **(page 722)** Migrate a config server to a new system while keeping the same hostname. This procedure requires changing the DNS entry to point to the new system.

Migrate Config Servers with Different Hostnames **(page 723)** Migrate a config server to a new system that uses a new hostname. If possible, avoid changing the hostname and instead use the *Migrate Config Servers with the Same Hostname* (page 722) procedure.

Replace Disabled Config Server **(page 724)** Replaces a config server that has become inoperable. This procedure assumes that the hostname does not change.

Migrate a Sharded Cluster to Different Hardware **(page 725)** Migrate a sharded cluster to a different hardware system, for example, when moving a pre-production environment to production.

Backup Cluster Metadata **(page 728)** Create a backup of a sharded cluster's metadata while keeping the cluster operational.

Configure Behavior of Balancer Process in Sharded Clusters **(page 728)** Manage the balancer's behavior by scheduling a balancing window, changing size settings, or requiring replication before migration.

Manage Sharded Cluster Balancer **(page 730)** View balancer status and manage balancer behavior.

Remove Shards from an Existing Sharded Cluster **(page 734)** Migrate a single shard's data and remove the shard.

View Cluster Configuration

On this page

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the *Config Database* (page 754). A database has sharding enabled if the value of the `partitioned` field is `true`. Connect to a `mongos` instance with a `mongo` shell, and run the following operation to get a full list of databases with sharding enabled:

```
use config
db.databases.find( { "partitioned": true } )
```

Example

You can use the following sequence of commands when to return a list of all databases in the cluster:

```
use config
db.databases.find()
```

If this returns the following result set:

```
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "animals", "partitioned" : true, "primary" : "m0.example.net:30001" }
{ "_id" : "farms", "partitioned" : false, "primary" : "m1.example2.net:27017" }
```

Then sharding is only enabled for the `animals` database.

List Shards

To list the current set of configured shards, use the `listShards` command, as follows:

```
use admin
db.runCommand( { listShards : 1 } )
```

View Cluster Details

To view cluster details, issue `db.printShardingStatus()` or `sh.status()`. Both methods return the same output.

Example

In the following example output from `sh.status()`

- `sharding version` displays the version number of the shard metadata.
- `shards` displays a list of the `mongod` instances used as shards in the cluster.
- `databases` displays all databases in the cluster, including database that do not have sharding enabled.
- The `chunks` information for the `foo` database displays how many chunks are on each shard and displays the range of each chunk.

```
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000",  "host" : "m0.example.net:30001" }
    {  "_id" : "shard0001",  "host" : "m3.example2.net:50000" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "contacts",  "partitioned" : true,  "primary" : "shard0000" }
        foo.contacts
            shard key: { "zip" : 1 }
            chunks:
                shard0001    2
                shard0002    3
                shard0000    2
            { "zip" : { "$minKey" : 1 } } -->> { "zip" : "56000" } on : shard0001 { "t" : 2, "i" : 0
            { "zip" : 56000 } -->> { "zip" : "56800" } on : shard0002 { "t" : 3, "i" : 4 }
            { "zip" : 56800 } -->> { "zip" : "57088" } on : shard0002 { "t" : 4, "i" : 2 }
            { "zip" : 57088 } -->> { "zip" : "57500" } on : shard0002 { "t" : 4, "i" : 3 }
            { "zip" : 57500 } -->> { "zip" : "58140" } on : shard0001 { "t" : 4, "i" : 0 }
            { "zip" : 58140 } -->> { "zip" : "59000" } on : shard0000 { "t" : 4, "i" : 1 }
            { "zip" : 59000 } -->> { "zip" : { "$maxKey" : 1 } } on : shard0000 { "t" : 3, "i" : 3 }
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
```

Migrate Config Servers with the Same Hostname

This procedure migrates a *config server* (page 684) in a *sharded cluster* (page 681) to a new system that uses *the same* hostname.

To migrate all the config servers in a cluster, perform this procedure for each config server separately and migrate the config servers in reverse order from how they are listed in the `mongos` instances' `configDB` string. Start with the last config server listed in the `configDB` string.

1. Shut down the config server.

   This renders all config data for the sharded cluster "read only."

2. Change the DNS entry that points to the system that provided the old config server, so that the *same* hostname points to the new system. How you do this depends on how you organize your DNS and hostname resolution services.

3. Copy the contents of `dbPath` from the old config server to the new config server.

   For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, you might issue a command similar to the following:

   ```
   rsync -az /data/configdb/ mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

When you start the third config server, your cluster will become writable and it will be able to create new splits and migrate chunks as needed.

Migrate Config Servers with Different Hostnames

On this page

Overview

Sharded clusters use a group of three config servers to store cluster meta data, and all three config servers must be available to support cluster metadata changes that include chunk splits and migrations. If one of the config servers is unavailable or inoperable, you must replace it as soon as possible.

This procedure migrates a *config server* (page 684) in a *sharded cluster* (page 681) 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 722).

Considerations

Changing a *config server's* (page 684) hostname **requires downtime** and requires restarting every process in the sharded cluster.

While migrating config servers, always make sure that all `mongos` instances have three config servers specified in the `configDB` setting at all times. Also ensure that you specify the config servers in the same order for each `mongos` instance's `configDB` setting.

Procedure

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 732) for more information.

2. Shut down the config server to migrate.

   This renders all config data for the sharded cluster "read only."

3. Copy the contents of `dbPath` from the old config server to the new config server. For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, use a command that resembles the following:

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

5. Shut down all existing MongoDB processes. This includes:
   - the `mongod` instances for the shards.
   - the `mongod` instances for the existing *config databases* (page 754).
   - the `mongos` instances.
6. Restart all shard `mongod` instances.
7. Restart the `mongod` instances for the two existing non-migrated config servers.
8. Update the `configDB` setting for each `mongos` instances.
9. Restart the `mongos` instances.
10. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 732) section for more information on managing the balancer process.

Replace Disabled Config Server

On this page

Overview

Sharded clusters use a group of three config servers to store cluster meta data, and all three config servers must be available to support cluster metadata changes that include chunk splits and migrations. If one of the config servers is unavailable or inoperable you must replace it as soon as possible.

This procedure replaces an inoperable *config server* (page 684) in a *sharded cluster* (page 681). 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 723).

Considerations

In the course of this procedure *never* remove a config server from the `configDB` parameter on any of the `mongos` instances.

Procedure

Step 1: Provision a new system, with the same IP address and hostname as the previous host. You will have to ensure the new system has the same IP address and hostname as the system it's replacing *or* you will need to modify the DNS records and wait for them to propagate.

Step 2: Shut down *one* of the remaining config servers. Copy all of this host's `dbPath` path from the current system to the system that will provide the new config server. This command, issued on the system with the data files, may resemble the following:

```
rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
```

Step 3: If necessary, update DNS and/or networking. Ensure the new config server is accessible by the same name as the previous config server.

Step 4: Start the *new* config server.

```
mongod --configsvr
```

Migrate a Sharded Cluster to Different Hardware

On this page

This procedure moves the components of the *sharded cluster* to a new hardware system without downtime for reads and writes.

Important: While the migration is in progress, do not attempt to change to the *cluster metadata* (page 703). Do not use any operation that modifies the cluster metadata *in any way*. For example, do not create or drop databases, create or drop collections, or use any sharding commands.

If your cluster includes a shard backed by a *standalone* `mongod` instance, consider *converting the standalone to a replica set* (page 619) to simplify migration and to let you keep the cluster online during future maintenance. Migrating a shard as standalone is a multi-step process that may require downtime.

To migrate a cluster to new hardware, perform the following tasks.

Disable the Balancer

Disable the balancer to stop *chunk migration* (page 700) and do not perform any metadata write operations until the process finishes. If a migration is in progress, the balancer will complete the in-progress migration before stopping.

To disable the balancer, connect to one of the cluster's `mongos` instances and issue the following method:

```
sh.stopBalancer()
```

To check the balancer state, issue the `sh.getBalancerState()` method.

For more information, see *Disable the Balancer* (page 732).

Migrate Each Config Server Separately

Migrate each *config server* (page 684) by starting with the *last* config server listed in the `configDB` string. Proceed in reverse order of the `configDB` string. Migrate and restart a config server before proceeding to the next. Do not rename a config server during this process.

Note: If the name or address that a sharded cluster uses to connect to a config server changes, you must restart **every** `mongod` and `mongos` instance in the sharded cluster. Avoid downtime by using CNAMEs to identify config servers within the MongoDB deployment.

See *Migrate Config Servers with Different Hostnames* (page 723) for more information.

Important: Start with the *last* config server listed in `configDB`.

---

1. Shut down the config server.

   This renders all config data for the sharded cluster "read only."

2. Change the DNS entry that points to the system that provided the old config server, so that the *same* hostname points to the new system. How you do this depends on how you organize your DNS and hostname resolution services.

3. Copy the contents of `dbPath` from the old config server to the new config server.

   For example, to copy the contents of `dbPath` to a machine named `mongodb.config2.example.net`, you might issue a command similar to the following:

   ```
   rsync -az /data/configdb/ mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

Restart the `mongos` Instances

If the `configDB` string will change as part of the migration, you must shut down *all* `mongos` instances before changing the `configDB` string. This avoids errors in the sharded cluster over `configDB` string conflicts.

If the `configDB` string will remain the same, you can migrate the `mongos` instances sequentially or all at once.

1. Shut down the `mongos` instances using the `shutdown` command. If the `configDB` string is changing, shut down *all* `mongos` instances.

2. If the hostname has changed for any of the config servers, update the `configDB` string for each `mongos` instance. The `mongos` instances must all use the same `configDB` string. The strings must list identical host names in identical order.

   Tip

   To avoid downtime, give each config server a logical DNS name (unrelated to the server's physical or virtual hostname). Without logical DNS names, moving or renaming a config server requires shutting down every `mongod` and `mongos` instance in the sharded cluster.

   ---

3. Restart the `mongos` instances being sure to use the updated `configDB` string if hostnames have changed.

For more information, see *Start the mongos Instances* (page 706).

Migrate the Shards

Migrate the shards one at a time. For each shard, follow the appropriate procedure in this section.

Migrate a Replica Set Shard To migrate a sharded cluster, migrate each member separately. First migrate the non-primary members, and then migrate the *primary* last.

If the replica set has two voting members, add an *arbiter* (page 574) to the replica set to ensure the set keeps a majority of its votes available during the migration. You can remove the arbiter after completing the migration.

Migrate a Member of a Replica Set Shard

1. Shut down the `mongod` process. To ensure a clean shutdown, use the `shutdown` command.
2. Move the data directory (i.e., the `dbPath`) to the new machine.
3. Restart the `mongod` process at the new location.
4. Connect to the replica set's current primary.
5. If the hostname of the member has changed, use `rs.reconfig()` to update the *replica set configuration document* (page 659) with the new hostname.

   For example, the following sequence of commands updates the hostname for the instance at position `2` in the `members` array:

   ```
   cfg = rs.conf()
   cfg.members[2].host = "pocatello.example.net:27017"
   rs.reconfig(cfg)
   ```

   For more information on updating the configuration document, see *replica-set-reconfiguration-usage*.
6. To confirm the new configuration, issue `rs.conf()`.
7. Wait for the member to recover. To check the member's state, issue `rs.status()`.

Migrate the Primary in a Replica Set Shard While migrating the replica set's primary, the set must elect a new primary. This failover process which renders the replica set unavailable to perform reads or accept writes for the duration of the election, which typically completes quickly. If possible, plan the migration during a maintenance window.

1. Step down the primary to allow the normal *failover* (page 583) process. To step down the primary, connect to the primary and issue the either the `replSetStepDown` command or the `rs.stepDown()` method. The following example shows the `rs.stepDown()` method:

   ```
   rs.stepDown()
   ```

2. Once the primary has stepped down and another member has become `PRIMARY` (page 667) state. To migrate the stepped-down primary, follow the *Migrate a Member of a Replica Set Shard* (page 727) procedure

   You can check the output of `rs.status()` to confirm the change in status.

Migrate a Standalone Shard The ideal procedure for migrating a standalone shard is to *convert the standalone to a replica set* (page 619) and then use the procedure for *migrating a replica set shard* (page 727). In production clusters, all shards should be replica sets, which provides continued availability during maintenance windows.

Migrating a shard as standalone is a multi-step process during which part of the shard may be unavailable. If the shard is the *primary shard* for a database,the process includes the `movePrimary` command. While the `movePrimary` runs, you should stop modifying data in that database. To migrate the standalone shard, use the *Remove Shards from an Existing Sharded Cluster* (page 734) procedure.

Re-Enable the Balancer

To complete the migration, re-enable the balancer to resume *chunk migrations* (page 700).

Connect to one of the cluster's `mongos` instances and pass `true` to the `sh.setBalancerState()` method:

```
sh.setBalancerState(true)
```

To check the balancer state, issue the `sh.getBalancerState()` method.

For more information, see *Enable the Balancer* (page 733).

Backup Cluster Metadata

This procedure shuts down the `mongod` instance of a *config server* (page 684) in order to create a backup of a *sharded cluster's* (page 675) 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 [14].

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 732) for more information.
2. Shut down one of the config databases.
3. Create a full copy of the data files (i.e. the path specified by the `dbPath` option for the config instance.)
4. Restart the original configuration server.
5. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 732) section for more information on managing the balancer process.

See also:

MongoDB Backup Methods (page 192).

Configure Behavior of Balancer Process in Sharded Clusters

On this page

[14] 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 684) for more information.

The balancer is a process that runs on *one* of the `mongos` instances in a cluster and ensures that *chunks* are evenly distributed throughout a sharded cluster. In most deployments, the default balancer configuration is sufficient for normal operation. However, administrators might need to modify balancer behavior depending on application or operational requirements. If you encounter a situation where you need to modify the behavior of the balancer, use the procedures described in this document.

For conceptual information about the balancer, see *Sharded Collection Balancing* (page 698) and *Cluster Balancer* (page 698).

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 731)
- *Remove a Balancing Window Schedule* (page 732).

The `mongos` instances use their own local timezones when respecting balancer window.

Configure Default Chunk Size

The default chunk size for a sharded cluster is 64 megabytes. In most situations, the default size is appropriate for splitting and migrating chunks. For information on how chunk size affects deployments, see details, see *Chunk Size* (page 702).

Changing the default chunk size affects chunks that are processes during migrations and auto-splits but does not retroactively affect all chunks.

To configure default chunk size, see *Modify Chunk Size in a Sharded Cluster* (page 743).

Change the Maximum Storage Size for a Given Shard

The `maxSize` field in the `shards` (page 759) collection in the *config database* (page 754) sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If `mapped` size [15] is above a shard's `maxSize`, the balancer will not move chunks to the shard. Also, the balancer will not move chunks off an overloaded shard. This must happen manually. The `maxSize` value only affects the balancer's selection of destination shards.

By default, `maxSize` is not specified, allowing shards to consume the total amount of available space on their machines if necessary.

You can set `maxSize` both when adding a shard and once a shard is running.

To set `maxSize` when adding a shard, set the `addShard` command's `maxSize` parameter to the maximum size in megabytes. For example, the following command run in the `mongo` shell adds a shard with a maximum size of 125 megabytes:

```
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 759) collection in the *config database* (page 754). Set the `maxSize` in megabytes.

Example

[15] This value includes the mapped size of all data files including the‘‘local‘‘ and `admin` databases. Account for this when setting `maxSize`.

Assume you have the following shard without a `maxSize` field:

```
{ "_id" : "shard0000", "host" : "example.net:34001" }
```

Run the following sequence of commands in the `mongo` shell to insert a `maxSize` of 125 megabytes:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 125 } } )
```

To later increase the `maxSize` setting to 250 megabytes, run the following:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 250 } } )
```

Change Replication Behavior for Chunk Migration (Secondary Throttle)

The `_secondaryThrottle` parameter of the balancer and the `moveChunk` command affects the replication behavior during *chunk migration* (page 701). By default, `_secondaryThrottle` is `true`, which means each document move during chunk migration propagates to at least one secondary before the balancer proceeds with its next operation. For more information on the replication behavior during various steps of chunk migration, see *Chunk Migration and Replication* (page 701).

To change the balancer's `_secondaryThrottle` value, connect to a `mongos` instance and directly update the `_secondaryThrottle` value in the `settings` (page 758) collection of the *config database* (page 754). For example, from a `mongo` shell connected to a `mongos`, issue the following command:

```
use config
db.settings.update(
   { "_id" : "balancer" },
   { $set : { "_secondaryThrottle" : false } },
   { upsert : true }
)
```

The effects of changing the `_secondaryThrottle` value may not be immediate. To ensure an immediate effect, stop and restart the balancer to enable the selected value of `_secondaryThrottle`. See *Manage Sharded Cluster Balancer* (page 730) for details.

Manage Sharded Cluster Balancer

On this page

This page describes common administrative procedures related to balancing. For an introduction to balancing, see *Sharded Collection Balancing* (page 698). For lower level information on balancing, see *Cluster Balancer* (page 698).

See also:

Configure Behavior of Balancer Process in Sharded Clusters (page 728)

Check the Balancer State

The following command checks if the balancer is enabled (i.e. that the balancer is allowed to run). The command does not check if the balancer is active (i.e. if it is actively balancing chunks).

To see if the balancer is enabled in your *cluster*, issue the following command, which returns a boolean:

```
sh.getBalancerState()
```

Check the Balancer Lock

To see if the balancer process is active in your *cluster*, do the following:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database* (page 754):

   ```
   use config
   ```

3. Use the following query to return the balancer lock:

   ```
   db.locks.find( { _id : "balancer" } ).pretty()
   ```

When this command returns, you will see output like the following:

```
{   "_id" : "balancer",
"process" : "mongos0.example.net:1292810611:1804289383",
  "state" : 2,
     "ts" : ObjectId("4d0f872630c42d1978be8a2e"),
   "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)",
    "who" : "mongos0.example.net:1292810611:1804289383:Balancer:846930886",
    "why" : "doing balance round" }
```

This output confirms that:

- The balancer originates from the `mongos` running on the system with the hostname `mongos0.example.net`.
- The value in the `state` field indicates that a `mongos` has the lock. For version 2.0 and later, the value of an active lock is `2`; for earlier versions the value is `1`.

Schedule the Balancing Window

In some situations, particularly when your data set grows slowly and a migration can impact performance, it's useful to be able to ensure that the balancer is active only at certain times. Use the following procedure to specify a window during which the *balancer* will be able to migrate chunks:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database* (page 754):

   ```
   use config
   ```

3. Issue the following operation to ensure the balancer is not in the `stopped` state:

```
sh.setBalancerState( true )
```

The balancer will not activate if in the `stopped` state or outside the `activeWindow` timeframe.

4. Use an operation modeled on the following `update()` operation to modify the balancer's window:

```
db.settings.update(
   { _id: "balancer" },
   { $set: { activeWindow : { start : "<start-time>", stop : "<stop-time>" } } },
   { upsert: true }
)
```

Replace `<start-time>` and `<end-time>` with time values using two digit hour and minute values (i.e. `HH:MM`) that specify the beginning and end boundaries of the balancing window.

- For `HH` values, use hour values ranging from `00` - `23`.
- For `MM` value, use minute values ranging from `00` - `59`.

The start and stop times will be evaluated relative to the time zone of each individual `mongos` instance in the sharded cluster. If your `mongos` instances are physically located in different time zones, use a common time zone (e.g. GMT) to ensure that the balancer window is interpreted correctly.

For instance, running the following will force the balancer to run between 11PM and 6AM local time only:

```
db.settings.update(
   { _id: "balancer" },
   { $set: { activeWindow : { start: "23:00", stop: "6:00" } } },
   { upsert: true }
)
```

Note: The balancer window must be sufficient to *complete* the migration of all data inserted during the day.

As data insert rates can change based on activity and usage patterns, it is important to ensure that the balancing window you select will be sufficient to support the needs of your deployment.

Do not use the `sh.startBalancer()` method when you have set an `activeWindow`.

Remove a Balancing Window Schedule

If you have *set the balancing window* (page 731) and wish to remove the schedule so that the balancer is always running, issue the following sequence of operations:

```
use config
db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

Disable the Balancer

By default the balancer may run at any time and only moves chunks as needed. To disable the balancer for a short period of time and prevent all migration, use the following procedure:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following operation to disable the balancer:

```
sh.stopBalancer()
```

If a migration is in progress, the system will complete the in-progress migration before stopping.

3. To verify that the balancer will not start, issue the following command, which returns `false` if the balancer is disabled:

   ```
   sh.getBalancerState()
   ```

   Optionally, to verify no migrations are in progress after disabling, issue the following operation in the `mongo` shell:

   ```
   use config
   while( sh.isBalancerRunning() ) {
             print("waiting...");
             sleep(1000);
   }
   ```

Note: To disable the balancer from a driver that does not have the `sh.stopBalancer()` or `sh.setBalancerState()` helpers, issue the following command from the `config` database:

```
db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } ,  { upsert: true } )
```

Enable the Balancer

Use this procedure if you have disabled the balancer and are ready to re-enable it:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue one of the following operations to enable the balancer:

   From the `mongo` shell, issue:

   ```
   sh.setBalancerState(true)
   ```

   From a driver that does not have the `sh.startBalancer()` helper, issue the following from the `config` database:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , { upsert: true } )
   ```

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 192), 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 731) 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 732) for the duration of the backup procedure.

If you turn the balancer off while it is in the middle of a balancing round, the shut down is not instantaneous. The balancer completes the chunk move in-progress and then ceases all further balancing rounds.

Before starting a backup operation, confirm that the balancer is not active. You can use the following command to determine if the balancer is active:

```
!sh.getBalancerState() && !sh.isBalancerRunning()
```

When the backup procedure is complete you can reactivate the balancer process.

Disable Balancing on a Collection

You can disable balancing for a specific collection with the `sh.disableBalancing()` method. You may want to disable the balancer for a specific collection to support maintenance operations or atypical workloads, for example, during data ingestions or data exports.

When you disable balancing on a collection, MongoDB will not interrupt in progress migrations.

To disable balancing on a collection, connect to a `mongos` with the `mongo` shell and call the `sh.disableBalancing()` method.

For example:

```
sh.disableBalancing("students.grades")
```

The `sh.disableBalancing()` method accepts as its parameter the full *namespace* of the collection.

Enable Balancing on a Collection

You can enable balancing for a specific collection with the `sh.enableBalancing()` method.

When you enable balancing for a collection, MongoDB will not *immediately* begin balancing data. However, if the data in your sharded collection is not balanced, MongoDB will be able to begin distributing the data more evenly.

To enable balancing on a collection, connect to a `mongos` with the `mongo` shell and call the `sh.enableBalancing()` method.

For example:

```
sh.enableBalancing("students.grades")
```

The `sh.enableBalancing()` method accepts as its parameter the full *namespace* of the collection.

Confirm Balancing is Enabled or Disabled

To confirm whether balancing for a collection is enabled or disabled, query the `collections` collection in the `config` database for the collection *namespace* and check the `noBalance` field. For example:

```
db.getSiblingDB("config").collections.findOne({_id : "students.grades"}).noBalance;
```

This operation will return a null error, `true`, `false`, or no output:

- A null error indicates the collection namespace is incorrect.
- If the result is `true`, balancing is disabled.
- If the result is `false`, balancing is enabled currently but has been disabled in the past for the collection. Balancing of this collection will begin the next time the balancer runs.
- If the operation returns no output, balancing is enabled currently and has never been disabled in the past for this collection. Balancing of this collection will begin the next time the balancer runs.

Remove Shards from an Existing Sharded Cluster

On this page

To remove a *shard* you must ensure the shard's data is migrated to the remaining shards in the cluster. This procedure describes how to safely migrate data and how to remove a shard.

This procedure describes how to safely remove a *single* shard. *Do not* use this procedure to migrate an entire cluster to new hardware. To migrate an entire shard to new hardware, migrate individual shards as if they were independent replica sets.

To remove a shard, first connect to one of the cluster's `mongos` instances using `mongo` shell. Then use the sequence of tasks in this document to remove a shard from the cluster.

Ensure the Balancer Process is Enabled

To successfully migrate data from a shard, the *balancer* process **must** be enabled. Check the balancer state using the `sh.getBalancerState()` helper in the `mongo` shell. For more information, see the section on *balancer operations* (page 732).

Determine the Name of the Shard to Remove

To determine the name of the shard, connect to a `mongos` instance with the `mongo` shell and either:

- Use the `listShards` command, as in the following:

  ```
  db.adminCommand( { listShards: 1 } )
  ```

- Run either the `sh.status()` or the `db.printShardingStatus()` method.

The `shards._id` field lists the name of each shard.

Remove Chunks from the Shard

From the `admin` database, run the `removeShard` command. This begins "draining" chunks from the shard you are removing to other shards in the cluster. For example, for a shard named `mongodb0`, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

This operation returns immediately, with the following response:

```
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "mongodb0",
    "ok" : 1
}
```

Depending on your network capacity and the amount of data, this operation can take from a few minutes to several days to complete.

Check the Status of the Migration

To check the progress of the migration at any stage in the process, run `removeShard` from the `admin` database again. For example, for a shard named `mongodb0`, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

The command returns output similar to the following:

```
{
     "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : 42,
        "dbs" : 1
    },
    "ok" : 1
}
```

In the output, the `remaining` document displays the remaining number of chunks that MongoDB must migrate to other shards and the number of MongoDB databases that have "primary" status on this shard.

Continue checking the status of the *removeShard* command until the number of chunks remaining is `0`. Always run the command on the `admin` database. If you are on a database other than `admin`, you can use `sh._adminCommand` to run the command on `admin`.

Move Unsharded Data

If the shard is the *primary shard* for one or more databases in the cluster, then the shard will have unsharded data. If the shard is not the primary shard for any databases, skip to the next task, *Finalize the Migration* (page 737).

In a cluster, a database with unsharded collections stores those collections only on a single shard. That shard becomes the primary shard for that database. (Different databases in a cluster can have different primary shards.)

Warning: Do not perform this procedure until you have finished draining the shard.

1. To determine if the shard you are removing is the primary shard for any of the cluster's databases, issue one of the following methods:

   - `sh.status()`
   - `db.printShardingStatus()`

   In the resulting document, the `databases` field lists each database and its primary shard. For example, the following `database` field shows that the `products` database uses `mongodb0` as the primary shard:

   ```
   {  "_id" : "products",  "partitioned" : true,  "primary" : "mongodb0" }
   ```

2. To move a database to another shard, use the `movePrimary` command. For example, to migrate all remaining unsharded data from `mongodb0` to `mongodb1`, issue the following command:

   ```
   db.runCommand( { movePrimary: "products", to: "mongodb1" })
   ```

This command does not return until MongoDB completes moving all data, which may take a long time. The response from this command will resemble the following:

```
{ "primary" : "mongodb1", "ok" : 1 }
```

Finalize the Migration

To clean up all metadata information and finalize the removal, run `removeShard` again. For example, for a shard named `mongodb0`, run:

```
use admin
db.runCommand( { removeShard: "mongodb0" } )
```

A success message appears at completion:

```
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "mongodb0",
    "ok" : 1
}
```

Once the value of the `state` field is "completed", you may safely stop the processes comprising the `mongodb0` shard.

See also:

Backup and Restore Sharded Clusters (page 265)

10.3.3 Sharded Cluster Data Management

The following documents provide information in managing data in sharded clusters.

Create Chunks in a Sharded Cluster **(page 738)** Create chunks, or *pre-split* empty collection to ensure an even distribution of chunks during data ingestion.

Split Chunks in a Sharded Cluster **(page 738)** Manually create chunks in a sharded collection.

Migrate Chunks in a Sharded Cluster **(page 739)** Manually migrate chunks without using the automatic balance process.

Merge Chunks in a Sharded Cluster **(page 740)** Use the `mergeChunks` to manually combine chunk ranges.

Modify Chunk Size in a Sharded Cluster **(page 743)** Modify the default chunk size in a sharded collection

Clear jumbo Flag **(page 744)** Clear *jumbo* flag from a shard.

Tag Aware Sharding **(page 746)** Tags associate specific ranges of *shard key* values with specific shards for use in managing deployment patterns.

Manage Shard Tags **(page 747)** Use tags to associate specific ranges of shard key values with specific shards.

Enforce Unique Keys for Sharded Collections **(page 749)** Ensure that a field is always unique in all collections in a sharded cluster.

Shard GridFS Data Store **(page 751)** Choose whether to shard GridFS data in a sharded collection.

Create Chunks in a Sharded Cluster

Pre-splitting the chunk ranges in an empty sharded collection allows clients to insert data into an already partitioned collection. In most situations a *sharded cluster* will create and distribute chunks automatically without user intervention. However, in a limited number of cases, MongoDB cannot create enough chunks or distribute data fast enough to support required throughput. For example:

- If you want to partition an existing data collection that resides on a single shard.
- If you want to ingest a large volume of data into a cluster that isn't balanced, or where the ingestion of data will lead to data imbalance. For example, monotonically increasing or decreasing shard keys insert all data into a single chunk.

These operations are resource intensive for several reasons:

- Chunk migration requires copying all the data in the chunk from one shard to another.
- MongoDB can migrate only a single chunk at a time.
- MongoDB creates splits only after an insert operation.

Warning: Only pre-split an empty collection. If a collection already has data, MongoDB automatically splits the collection's data when you enable sharding for the collection. Subsequent attempts to manually create splits can lead to unpredictable chunk ranges and sizes as well as inefficient or ineffective balancing behavior.

To create chunks manually, use the following procedure:

1. Split empty chunks in your collection by manually performing the `split` command on chunks.

   Example

   To create chunks for documents in the `myapp.users` collection using the `email` field as the *shard key*, use the following operation in the `mongo` shell:

   ```
   for ( var x=97; x<97+26; x++ ){
     for( var y=97; y<97+26; y+=6 ) {
       var prefix = String.fromCharCode(x) + String.fromCharCode(y);
       db.runCommand( { split : "myapp.users" , middle : { email : prefix } } );
     }
   }
   ```

   This assumes a collection size of 100 million documents.

   For information on the balancer and automatic distribution of chunks across shards, see *Cluster Balancer* (page 698) and *Chunk Migration* (page 700). For information on manually migrating chunks, see *Migrate Chunks in a Sharded Cluster* (page 739).

Split Chunks in a Sharded Cluster

Normally, MongoDB splits a *chunk* after an insert if the chunk exceeds the maximum *chunk size* (page 702). However, you may want to split chunks manually if:

- you have a large amount of data in your cluster and very few *chunks*, as is the case after deploying a cluster using existing data.
- you expect to add a large amount of data that would initially reside in a single chunk or shard. For example, you plan to insert a large amount of data with *shard key* values between `300` and `400`, *but* all values of your shard keys are between `250` and `500` are in a single chunk.

Note: New in version 2.6: MongoDB provides the `mergeChunks` command to combine contiguous chunk ranges into a single chunk. See *Merge Chunks in a Sharded Cluster* (page 740) for more information.

The *balancer* may migrate recently split chunks to a new shard immediately if `mongos` predicts future insertions will benefit from the move. The balancer does not distinguish between chunks split manually and those split automatically by the system.

Warning: Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

Use `sh.status()` to determine the current chunk ranges across the cluster.

To split chunks manually, use the `split` command with either fields `middle` or `find`. The `mongo` shell provides the helper methods `sh.splitFind()` and `sh.splitAt()`.

`splitFind()` splits the chunk that contains the *first* document returned that matches this query into two equally sized chunks. You must specify the full namespace (i.e. "`<database>.<collection>`") of the sharded collection to `splitFind()`. The query in `splitFind()` does not need to use the shard key, though it nearly always makes sense to do so.

Example

The following command splits the chunk that contains the value of `63109` for the `zipcode` field in the `people` collection of the `records` database:

```
sh.splitFind( "records.people", { "zipcode": "63109" } )
```

Use `splitAt()` to split a chunk in two, using the queried document as the lower bound in the new chunk:

Example

The following command splits the chunk that contains the value of `63109` for the `zipcode` field in the `people` collection of the `records` database.

```
sh.splitAt( "records.people", { "zipcode": "63109" } )
```

Note: `splitAt()` does not necessarily split the chunk into two equally sized chunks. The split occurs at the location of the document matching the query, regardless of where that document is in the chunk.

Migrate Chunks in a Sharded Cluster

In most circumstances, you should let the automatic *balancer* migrate *chunks* between *shards*. However, you may want to migrate chunks manually in a few cases:

- When *pre-splitting* an empty collection, migrate chunks manually to distribute them evenly across the shards. Use pre-splitting in limited situations to support bulk data ingestion.
- If the balancer in an active cluster cannot distribute chunks within the *balancing window* (page 731), then you will have to migrate chunks manually.

To manually migrate chunks, use the `moveChunk` command. For more information on how the automatic balancer moves chunks between shards, see *Cluster Balancer* (page 698) and *Chunk Migration* (page 700).

Example

Migrate a single chunk

The following example assumes that the field `username` is the *shard key* for a collection named `users` in the `myapp` database, and that the value `smith` exists within the *chunk* to migrate. Migrate the chunk using the following command in the `mongo` shell.

```
db.adminCommand( { moveChunk : "myapp.users",
                   find : {username : "smith"},
                   to : "mongodb-shard3.example.net" } )
```

This command moves the chunk that includes the shard key value "smith" to the *shard* named `mongodb-shard3.example.net`. The command will block until the migration is complete.

Tip

To return a list of shards, use the `listShards` command.

Example

Evenly migrate chunks

To evenly migrate chunks for the `myapp.users` collection, put each prefix chunk on the next shard from the other and run the following commands in the mongo shell:

```
var shServer = [ "sh0.example.net", "sh1.example.net", "sh2.example.net", "sh3.example.net", "sh4.exa
for ( var x=97; x<97+26; x++ ){
  for( var y=97; y<97+26; y+=6 ) {
    var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    db.adminCommand({moveChunk : "myapp.users", find : {email : prefix}, to : shServer[(y-97)/6]})
  }
}
```

See *Create Chunks in a Sharded Cluster* (page 738) for an introduction to pre-splitting.

New in version 2.2: The `moveChunk` command has the: `_secondaryThrottle` parameter. When set to `true`, MongoDB ensures that changes to shards as part of chunk migrations replicate to *secondaries* throughout the migration operation. For more information, see *Change Replication Behavior for Chunk Migration (Secondary Throttle)* (page 730).

Changed in version 2.4: In 2.4, `_secondaryThrottle` is `true` by default.

Warning: The `moveChunk` command may produce the following error message:

```
The collection's metadata lock is already taken.
```

This occurs when clients have too many open *cursors* that access the migrating chunk. You may either wait until the cursors complete their operations or close the cursors manually.

Merge Chunks in a Sharded Cluster

On this page

Overview

The `mergeChunks` command allows you to collapse empty chunks into neighboring chunks on the same shard. A *chunk* is empty if it has no documents associated with its shard key range.

Important: Empty *chunks* can make the *balancer* assess the cluster as properly balanced when it is not.

Empty chunks can occur under various circumstances, including:

- If a *pre-split* (page 738) creates too many chunks, the distribution of data to chunks may be uneven.
- If you delete many documents from a sharded collection, some chunks may no longer contain data.

This tutorial explains how to identify chunks available to merge, and how to merge those chunks with neighboring chunks.

Procedure

Note: Examples in this procedure use a `users` *collection* in the `test` *database*, using the `username` filed as a *shard key*.

Identify Chunk Ranges In the `mongo` shell, identify the *chunk* ranges with the following operation:

```
sh.status()
```

The output of the `sh.status()` will resemble the following:

```
--- Sharding Status ---
sharding version: {
     "_id" : 1,
     "version" : 4,
     "minCompatibleVersion" : 4,
     "currentVersion" : 5,
     "clusterId" : ObjectId("5260032c901f6712dcd8f400")
}
shards:
     {  "_id" : "shard0000",  "host" : "localhost:30000" }
     {  "_id" : "shard0001",  "host" : "localhost:30001" }
  databases:
     {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
     {  "_id" : "test",  "partitioned" : true,  "primary" : "shard0001" }
            test.users
                    shard key: { "username" : 1 }
                    chunks:
                            shard0000       7
                            shard0001       7
                    { "username" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard0
                    { "username" : "user16643" } -->> { "username" : "user2329" } on : shard0000 Tim
```

```
{ "username" : "user2329" } -->> { "username" : "user29937" } on : shard0000 Tir
{ "username" : "user29937" } -->> { "username" : "user36583" } on : shard0000 Ti
{ "username" : "user36583" } -->> { "username" : "user43229" } on : shard0000 Ti
{ "username" : "user43229" } -->> { "username" : "user49877" } on : shard0000 Ti
{ "username" : "user49877" } -->> { "username" : "user56522" } on : shard0000 Ti
{ "username" : "user56522" } -->> { "username" : "user63169" } on : shard0001 Ti
{ "username" : "user63169" } -->> { "username" : "user69816" } on : shard0001 Ti
{ "username" : "user69816" } -->> { "username" : "user76462" } on : shard0001 Ti
{ "username" : "user76462" } -->> { "username" : "user83108" } on : shard0001 Ti
{ "username" : "user83108" } -->> { "username" : "user89756" } on : shard0001 Ti
{ "username" : "user89756" } -->> { "username" : "user96401" } on : shard0001 Ti
{ "username" : "user96401" } -->> { "username" : { "$maxKey" : 1 } } on : shard0
```

The chunk ranges appear after the chunk counts for each sharded collection, as in the following excerpts:

Chunk counts:

```
chunks:
        shard0000       7
        shard0001       7
```

Chunk range:

```
{ "username" : "user36583" } -->> { "username" : "user43229" } on : shard0000 Timestamp(6, 0)
```

Verify a Chunk is Empty The `mergeChunks` command requires at least one empty input chunk. In the `mongo` shell, check the amount of data in a chunk using an operation that resembles:

```
db.runCommand({
   "dataSize": "test.users",
   "keyPattern": { username: 1 },
   "min": { "username": "user36583" },
   "max": { "username": "user43229" }
})
```

If the input chunk to `dataSize` is empty, `dataSize` produces output similar to:

```
{ "size" : 0, "numObjects" : 0, "millis" : 0, "ok" : 1 }
```

Merge Chunks Merge two contiguous *chunks* on the same *shard*, where at least one of the contains no data, with an operation that resembles the following:

```
db.runCommand( { mergeChunks: "test.users",
                 bounds: [ { "username": "user68982" },
                           { "username": "user95197" } ]
             } )
```

On success, `mergeChunks` produces the following output:

```
{ "ok" : 1 }
```

On any failure condition, `mergeChunks` returns a document where the value of the `ok` field is `0`.

View Merged Chunks Ranges After merging all empty chunks, confirm the new chunk, as follows:

```
sh.status()
```

The output of `sh.status()` should resemble:

```
--- Sharding Status ---
sharding version: {
     "_id" : 1,
     "version" : 4,
     "minCompatibleVersion" : 4,
     "currentVersion" : 5,
     "clusterId" : ObjectId("5260032c901f6712dcd8f400")
}
shards:
     {  "_id" : "shard0000",  "host" : "localhost:30000" }
     {  "_id" : "shard0001",  "host" : "localhost:30001" }
  databases:
     {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
     {  "_id" : "test",  "partitioned" : true,  "primary" : "shard0001" }
             test.users
                     shard key: { "username" : 1 }
                     chunks:
                             shard0000       2
                             shard0001       2
                     { "username" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard0
                     { "username" : "user16643" } -->> { "username" : "user56522" } on : shard0000 T
                     { "username" : "user56522" } -->> { "username" : "user96401" } on : shard0001 T
                     { "username" : "user96401" } -->> { "username" : { "$maxKey" : 1 } } on : shard0
```

Modify Chunk Size in a Sharded Cluster

When the first `mongos` connects to a set of *config servers*, it initializes the sharded cluster with a default chunk size of 64 megabytes. This default chunk size works well for most deployments; however, if you notice that automatic migrations have more I/O than your hardware can handle, you may want to reduce the chunk size. For automatic splits and migrations, a small chunk size leads to more rapid and frequent migrations. The allowed range of the chunk size is between 1 and 1024 megabytes, inclusive.

To modify the chunk size, use the following procedure:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *Config Database* (page 754):

   ```
   use config
   ```

3. Issue the following `save()` operation to store the global chunk size configuration value:

   ```
   db.settings.save( { _id:"chunksize", value: <sizeInMB> } )
   ```

Note: The `chunkSize` and `--chunkSize` options, passed at startup to the `mongos`, **do not** affect the chunk size after you have initialized the cluster.

To avoid confusion, *always* set the chunk size using the above procedure instead of the startup options.

Modifying the chunk size has several limitations:

- Automatic splitting only occurs on insert or update.
- If you lower the chunk size, it may take time for all chunks to split to the new size.
- Splits cannot be undone.

- If you increase the chunk size, existing chunks grow only through insertion or updates until they reach the new size.
- The allowed range of the chunk size is between 1 and 1024 megabytes, inclusive.

Clear `jumbo` Flag

On this page

If MongoDB cannot split a chunk that exceeds the *specified chunk size* (page 702) or contains a number of documents that exceeds the `max`, MongoDB labels the chunk as *jumbo* (page 702).

If the chunk size no longer hits the limits, MongoDB clears the `jumbo` flag for the chunk when the `mongos` reloads or rewrites the chunk metadata.

In cases where you need to clear the flag manually, the following procedures outline the steps to manually clear the `jumbo` flag.

Procedures

Divisible Chunks The preferred way to clear the `jumbo` flag from a chunk is to attempt to split the chunk. If the chunk is divisible, MongoDB removes the flag upon successful split of the chunk.

Step 1: Connect to `mongos`. Connect a `mongo` shell to a `mongos`.

Step 2: Find the `jumbo` Chunk. Run `sh.status(true)` to find the chunk labeled `jumbo`.

```
sh.status(true)
```

For example, the following output from sh.status(true) shows that chunk with shard key range `{ "x" : 2 } -->> { "x" : 4 }` is `jumbo`.

```
--- Sharding Status ---
  sharding version: {
     ...
  }
  shards:
     ...
  databases:
     ...
        test.foo
           shard key: { "x" : 1 }
        chunks:
             shard-b  2
             shard-a  2
        { "x" : { "$minKey" : 1 } } -->> { "x" : 1 } on : shard-b Timestamp(2, 0)
        { "x" : 1 } -->> { "x" : 2 } on : shard-a Timestamp(3, 1)
        { "x" : 2 } -->> { "x" : 4 } on : shard-a Timestamp(2, 2) jumbo
        { "x" : 4 } -->> { "x" : { "$maxKey" : 1 } } on : shard-b Timestamp(3, 0)
```

Step 3: Split the `jumbo` Chunk. Use either `sh.splitAt()` or `sh.splitFind()` to split the `jumbo` chunk.

```
sh.splitAt( "test.foo", { x: 3 })
```

MongoDB removes the `jumbo` flag upon successful split of the chunk.

Indivisible Chunks In some instances, MongoDB cannot split the no-longer `jumbo` chunk, such as a chunk with a range of single shard key value, and the preferred method to clear the flag is not applicable. In such cases, you can clear the flag using the following steps.

Important: Only use this method if the *preferred method* (page 744) is *not* applicable.

Before modifying the *config database* (page 754), *always* back up the config database.

If you clear the `jumbo` flag for a chunk that still exceeds the chunk size and/or the document number limit, MongoDB will re-label the chunk as `jumbo` when MongoDB tries to move the chunk.

Step 1: Stop the balancer. Disable the cluster balancer process temporarily, following the steps outlined in *Disable the Balancer* (page 732).

Step 2: Create a backup of `config` database. Use `mongodump` against a config server to create a backup of the `config` database. For example:

```
mongodump --db config --port <config server port> --out <output file>
```

Step 3: Connect to `mongos`. Connect a `mongo` shell to a `mongos`.

Step 4: Find the `jumbo` Chunk. Run `sh.status(true)` to find the chunk labeled `jumbo`.

```
sh.status(true)
```

For example, the following output from sh.status(true) shows that chunk with shard key range `{ "x" : 2 } -->> { "x" : 3 }` is `jumbo`.

```
--- Sharding Status ---
  sharding version: {
     ...
  }
  shards:
     ...
  databases:
     ...
        test.foo
           shard key: { "x" : 1 }
        chunks:
             shard-b  2
             shard-a  2
        { "x" : { "$minKey" : 1 } } -->> { "x" : 1 } on : shard-b Timestamp(2, 0)
        { "x" : 1 } -->> { "x" : 2 } on : shard-a Timestamp(3, 1)
        { "x" : 2 } -->> { "x" : 3 } on : shard-a Timestamp(2, 2) jumbo
        { "x" : 3 } -->> { "x" : { "$maxKey" : 1 } } on : shard-b Timestamp(3, 0)
```

Step 5: Update `chunks` collection. In the `chunks` collection of the `config` database, unset the `jumbo` flag for the chunk. For example,

```
db.getSiblingDB("config").chunks.update(
   { ns: "test.foo", min: { x: 2 }, jumbo: true },
   { $unset: { jumbo: "" } }
)
```

Step 6: Restart the balancer. Restart the balancer, following the steps in *Enable the Balancer* (page 733).

Step 7: Optional. Clear current cluster meta information. To ensure that `mongos` instances update their cluster information cache, run `flushRouterConfig` in the `admin` database.

```
db.adminCommand({ flushRouterConfig: 1 } )
```

Tag Aware Sharding

On this page

MongoDB supports tagging a range of *shard key* values to associate that range with a shard or group of shards. Those shards receive all inserts within the tagged range.

The balancer obeys tagged range associations, which enables the following deployment patterns:

- isolate a specific subset of data on a specific set of shards.
- ensure that the most relevant data reside on shards that are geographically closest to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

Considerations

- *Shard key* range tags are distinct from *replica set member tags* (page 594).
- *Hash-based sharding* only supports tag-aware sharding on entire collection.
- Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Behavior and Operations

The balancer migrates chunks of documents in a sharded collections to the shards associated with a tag that has a *shard key* range with an *upper* bound *greater* than the chunk's *lower* bound.

During balancing rounds, if the balancer detects that any chunks violate configured tags, the balancer migrates chunks in tagged ranges to shards associated with those tags.

After configuring tags with a shard key range, and associating it with a shard or shards, the cluster may take some time to balance the data among the shards. This depends on the division of chunks and the current distribution of data in the cluster.

Once configured, the balancer respects tag ranges during future *balancing rounds* (page 698).

See also:

Manage Shard Tags (page 747)

Additional Resource

MongoDB Multi-Data Center Deployments Whitepaper[16]

Manage Shard Tags

On this page

In a sharded cluster, you can use tags to associate specific ranges of a *shard key* with a specific *shard* or subset of shards.

Tag a Shard

Associate tags with a particular shard using the `sh.addShardTag()` method when connected to a `mongos` instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag `NYC` to two shards, and the tags `SFO` and `NRT` to a third shard:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

You may remove tags from a particular shard using the `sh.removeShardTag()` method when connected to a `mongos` instance, as in the following example, which removes the `NRT` tag from a shard:

```
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range

To assign a tag to a range of shard keys use the `sh.addTagRange()` method when connected to a `mongos` instance. Any given shard key range may only have *one* assigned tag. You cannot overlap defined ranges, or tag the same range more than once.

Example

[16] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs

Given a collection named `users` in the `records` database, sharded by the `zipcode` field. The following operations assign:

- two ranges of zip codes in Manhattan and Brooklyn the `NYC` tag
- one range of zip codes in San Francisco the `SFO` tag

```
sh.addTagRange("records.users", { zipcode: "10001" }, { zipcode: "10281" }, "NYC")
sh.addTagRange("records.users", { zipcode: "11201" }, { zipcode: "11240" }, "NYC")
sh.addTagRange("records.users", { zipcode: "94102" }, { zipcode: "94135" }, "SFO")
```

Note: Shard ranges are always inclusive of the lower value and exclusive of the upper boundary.

Remove a Tag From a Shard Key Range

The `mongod` does not provide a helper for removing a tag range. You may delete tag assignment from a shard key range by removing the corresponding document from the `tags` (page 759) collection of the `config` database.

Each document in the `tags` (page 759) holds the *namespace* of the sharded collection and a minimum shard key value.

Example

The following example removes the `NYC` tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

View Existing Shard Tags

The output from `sh.status()` lists tags associated with a shard, if any, for each shard. A shard's tags exist in the shard's document in the `shards` (page 759) 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 759) collection of the `config` database. The output of `sh.status()` displays all tag ranges. To return all shard key ranges tagged with `NYC`, use the following sequence of operations:

```
use config
db.tags.find({ tags: "NYC" })
```

Additional Resource

MongoDB Multi-Data Center Deployments Whitepaper[17]

[17] http://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs

Enforce Unique Keys for Sharded Collections

On this page

Overview

The `unique` constraint on indexes ensures that only one document can have a value for a field in a *collection*. For *sharded collections these unique indexes cannot enforce uniqueness* because insert and indexing operations are local to each shard.

MongoDB does not support creating new unique indexes in sharded collections and will not allow you to shard collections with unique indexes on fields other than the `_id` field.

If you need to ensure that a field is always unique in a sharded collection, there are three options:

1. Enforce uniqueness of the *shard key* (page 687).

   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 504).

2. Use a secondary collection to enforce uniqueness.

   Create a minimal collection that only contains the unique field and a reference to a document in the main collection. If you always insert into a secondary collection *before* inserting to the main collection, MongoDB will produce an error if you attempt to use a duplicate key.

   If you have a small data set, you may not need to shard this collection and you can create multiple unique indexes. Otherwise you can shard on a single unique key.

3. Use guaranteed unique identifiers.

   Universally unique identifiers (i.e. UUID) like the `ObjectId` are guaranteed to be unique.

Procedures

Unique Constraints on the Shard Key

Process To shard a collection using the `unique` constraint, specify the `shardCollection` command in the following form:

```
db.runCommand( { shardCollection : "test.users" , key : { email : 1 } , unique : true } );
```

Remember that the `_id` field index is always unique. By default, MongoDB inserts an `ObjectId` into the `_id` field. However, you can manually insert your own value into the `_id` field and use this as the shard key. To use the `_id` field as the shard key, use the following operation:

```
db.runCommand( { shardCollection : "test.users" } )
```

Limitations

- You can only enforce uniqueness on one single field in the collection using this method.
- If you use a compound shard key, you can only enforce uniqueness on the *combination* of component keys in the shard key.

In most cases, the best shard keys are compound keys that include elements that permit *write scaling* (page 689) and *query isolation* (page 690), as well as *high cardinality* (page 710). These ideal shard keys are not often the same keys that require uniqueness and enforcing unique values in these collections requires a different approach.

Unique Constraints on Arbitrary Fields If you cannot use a unique field as the shard key or if you need to enforce uniqueness over multiple fields, you must create another *collection* to act as a "proxy collection". This collection must contain both a reference to the original document (i.e. its `ObjectId`) and the unique key.

If you must shard this "proxy" collection, then shard on the unique key using the *above procedure* (page 749); 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*:

```
db = db.getSiblingDB('records');

var primary_id = ObjectId();

db.proxy.insert({
   "_id" : primary_id
   "email" : "example@example.net"
})

// if: the above operation returns successfully,
// then continue:

db.information.insert({
   "_id" : primary_id
   "email": "example@example.net"
   // additional information...
})
```

You must insert a document into the `proxy` collection first. If this operation succeeds, the `email` field is unique, and you may continue by inserting the actual document into the `information` collection.

See

The full documentation of: `ensureIndex()` and `shardCollection`.

Considerations

- Your application must catch errors when inserting documents into the "proxy" collection and must enforce consistency between the two collections.
- If the proxy collection requires sharding, you must shard on the single field on which you want to enforce uniqueness.
- To enforce uniqueness on more than one field using sharded proxy collections, you must have *one* proxy collection for *every* field for which to enforce uniqueness. If you create multiple unique indexes on a single proxy collection, you will *not* be able to shard proxy collections.

Use Guaranteed Unique Identifier The best way to ensure a field has unique values is to generate universally unique identifiers (UUID,) such as MongoDB's `ObjectId` values.

This approach is particularly useful for the''_id'' field, which *must* be unique: for collections where you are *not* sharding by the `_id` field the application is responsible for ensuring that the `_id` field is unique.

Shard GridFS Data Store

On this page

When sharding a *GridFS* store, consider the following:

`files` Collection

Most deployments will not need to shard the `files` collection. The `files` collection is typically small, and only contains metadata. None of the required keys for GridFS lend themselves to an even distribution in a sharded situation. If you *must* shard the `files` collection, use the `_id` field possibly in combination with an application field.

Leaving `files` unsharded means that all the file metadata documents live on one shard. For production GridFS stores you *must* store the `files` collection on a replica set.

`chunks` Collection

To shard the `chunks` collection by `{ files_id : 1 , n : 1 }`, issue commands similar to the following:

```
db.fs.chunks.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.

10.3.4 Troubleshoot Sharded Clusters

On this page

This section describes common strategies for troubleshooting *sharded cluster* deployments.

Config Database String Error

Start all `mongos` instances in a sharded cluster with an identical `configDB` string. If a `mongos` instance tries to connect to the sharded cluster with a `configDB` string that does not *exactly* match the string used by the other `mongos` instances, including the order of the hosts, the following errors occur:

```
could not initialize sharding on connection
```

And:

```
mongos specified a different config database string
```

To solve the issue, restart the `mongos` with the correct string.

Cursor Fails Because of Stale Config Data

A query returns the following warning when one or more of the `mongos` instances has not yet updated its cache of the cluster's metadata from the *config database*:

```
could not initialize cursor across all shards because : stale config detected
```

This warning *should* not propagate back to your application. The warning will repeat until all the `mongos` instances refresh their caches. To force an instance to refresh its cache, run the `flushRouterConfig` command.

Avoid Downtime when Moving Config Servers

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

10.4 Sharding Reference

On this page

10.4.1 Sharding Methods in the mongo Shell

Name	Description
sh._adminCommand	Runs a *database command* against the admin database, like db.runCommand(), but can confirm that it is issued against a mongos.
sh._checkFullName()	Tests a namespace to determine if its well formed.
sh._checkMongos()	Tests to see if the mongo shell is connected to a mongos instance.
sh._lastMigration()	Reports on the last *chunk* migration.
sh.addShard()	Adds a *shard* to a sharded cluster.
sh.addShardTag()	Associates a shard with a tag, to support *tag aware sharding* (page 746).
sh.addTagRange()	Associates range of shard keys with a shard tag, to support *tag aware sharding* (page 746).
sh.disableBalancing()	Disable balancing on a single collection in a sharded database. Does not affect balancing of other collections in a sharded cluster.
sh.enableBalancing()	Activates the sharded collection balancer process if previously disabled using sh.disableBalancing().
sh.enableSharding()	Enables sharding on a specific database.
sh.getBalancerHost()	Returns the name of a mongos that's responsible for the balancer process.
sh.getBalancerState()	Returns a boolean to report if the *balancer* is currently enabled.
sh.help()	Returns help text for the sh methods.
sh.isBalancerRunning()	Returns a boolean to report if the balancer process is currently migrating chunks.
sh.moveChunk()	Migrates a *chunk* in a *sharded cluster*.
sh.removeShardTag()	Removes the association between a shard and a shard tag.
sh.setBalancerState()	Enables or disables the *balancer* which migrates *chunks* between *shards*.
sh.shardCollection()	Enables sharding for a collection.
sh.splitAt()	Divides an existing *chunk* into two chunks using a specific value of the *shard key* as the dividing point.
sh.splitFind()	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.
sh.status()	Reports on the status of a *sharded cluster*, as db.printShardingStatus().
sh.stopBalancer()	Disables the *balancer* and waits for any in progress balancing rounds to complete.
sh.waitForBalancer()	Internal. Waits for the balancer state to change.
sh.waitForBalancerOff()	Internal. Waits until the balancer stops running.
sh.waitForDLock()	Internal. Waits for a specified distributed *sharded cluster* lock.
sh.waitForPingChange()	Internal. Waits for a change in ping state from one of the mongos in the sharded cluster.

10.4.2 Sharding Database Commands

The following database commands support *sharded clusters*.

Name	Description
`flushRouterConfig`	Forces an update to the cluster metadata cached by a `mongos`.
`addShard`	Adds a *shard* to a *sharded cluster*.
`cleanupOrphaned`	Removes orphaned data with shard key values outside of the ranges of the chunks owned by a shard.
`checkShardingIndex`	Internal command that validates index on shard key.
`enableSharding`	Enables sharding on a specific database.
`listShards`	Returns a list of configured shards.
`removeShard`	Starts the process of removing a shard from a sharded cluster.
`getShardMap`	Internal command that reports on the state of a sharded cluster.
`getShardVersion`	Internal command that returns the *config server* version.
`mergeChunks`	Provides the ability to combine chunks on a single shard.
`setShardVersion`	Internal command to sets the *config server* version.
`shardCollection`	Enables the sharding functionality for a collection, allowing the collection to be sharded.
`shardingState`	Reports whether the `mongod` is a member of a sharded cluster.
`unsetSharding`	Internal command that affects connections between instances in a MongoDB deployment.
`split`	Creates a new *chunk*.
`splitChunk`	Internal command to split chunk. Instead use the methods `sh.splitFind()` and `sh.splitAt()`.
`splitVector`	Internal command that determines split points.
`medianKey`	Deprecated internal command. See `splitVector`.
`moveChunk`	Internal command that migrates chunks between shards.
`movePrimary`	Reassigns the *primary shard* when removing a shard from a sharded cluster.
`isdbgrid`	Verifies that a process is a `mongos`.

10.4.3 Reference Documentation

Config Database **(page 754)** Complete documentation of the content of the `local` database that MongoDB uses to store sharded cluster metadata.

Config Database

On this page

The `config` database supports *sharded cluster* operation. See the *Sharding* (page 675) section of this manual for full documentation of sharded clusters.

Important: Consider the schema of the `config` database *internal* and may change between releases of MongoDB. The `config` database is not a dependable API, and users should not write data to the `config` database in the course of normal operation or maintenance.

Warning: Modification of the `config` database on a functioning system may lead to instability or inconsistent data sets. If you must modify the `config` database, use `mongodump` to create a full backup of the `config` database.

To access the `config` database, connect to a `mongos` instance in a sharded cluster, and use the following helper:

```
use config
```

You can return a list of the collections, with the following helper:

```
show collections
```

Collections

config

config.**changelog**

Internal MongoDB Metadata

The config (page 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The changelog (page 755) 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 755) 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 755) collection contains the following fields:

config.changelog.**_id**
The value of changelog._id is: <hostname>-<timestamp>-<increment>.

config.changelog.**server**
The hostname of the server that holds this data.

config.changelog.**clientAddr**
A string that holds the address of the client, a mongos instance that initiates this change.

config.changelog.**time**
A *ISODate* timestamp that reflects when the change occurred.

config.changelog.**what**
Reflects the type of change recorded. Possible values are:

- dropCollection
- dropCollection.start
- dropDatabase
- dropDatabase.start
- moveChunk.start
- moveChunk.commit
- split
- multi-split

config.changelog.**ns**
Namespace where the change occurred.

config.changelog.**details**
A *document* that contains additional details regarding the change. The structure of the details (page 756) document depends on the type of change.

config.**chunks**

Internal MongoDB Metadata

The config (page 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The chunks (page 756) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `collections` (page 757) 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 757) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `databases` (page 757) collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` (page 757) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The lockpings (page 757) collection keeps track of the active components in the sharded cluster. Given a cluster with a mongos running on example.com:30000, the document in the lockpings (page 757) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The locks (page 758) collection stores a distributed lock. This ensures that only one mongos instance can perform administrative tasks on the cluster at once. The mongos acting as *balancer* takes a lock by inserting a document resembling the following into the locks collection.

```
{
    "_id" : "balancer",
    "process" : "example.net:40000:1350402818:16807",
    "state" : 2,
    "ts" : ObjectId("507daeedf40e1879df62e5f3"),
    "when" : ISODate("2012-10-16T19:01:01.593Z"),
    "who" : "example.net:40000:1350402818:16807:Balancer:282475249",
    "why" : "doing balance round"
}
```

If a mongos holds the balancer lock, the state field has a value of 2, which means that balancer is active. The when field indicates when the balancer began the current operation.

Changed in version 2.0: The value of the state field was 1 before MongoDB 2.0.

config.mongos

Internal MongoDB Metadata

The config (page 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The mongos (page 758) collection stores a document for each mongos instance affiliated with the cluster. mongos instances send pings to all members of the cluster every 30 seconds so the cluster can verify that the mongos is active. The ping field shows the time of the last ping, while the up field reports the uptime of the mongos as of the last ping. The cluster maintains this collection for reporting purposes.

The following document shows the status of the mongos running on example.com:30000.

```
{ "_id" : "example.com:30000", "ping" : ISODate("2012-10-12T17:08:13.538Z"), "up" : 13699, "wait
```

config.settings

Internal MongoDB Metadata

The config (page 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The settings (page 758) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see *Modify Chunk Size in a Sharded Cluster* (page 743).
- Balancer status. To change status, see *Disable the Balancer* (page 732).

The following is an example settings collection:

```
{ "_id" : "chunksize", "value" : 64 }
{ "_id" : "balancer", "stopped" : false }
```

config.shards

Internal MongoDB Metadata

The config (page 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The shards (page 759) 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 746) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The tags (page 759) collection holds documents for each tagged shard key range in the cluster. The documents in the tags (page 759) 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 755) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The version (page 759) collection holds the current metadata version number. This collection contains only one document:

To access the version (page 759) collection you must use the db.getCollection() method. For example, to display the collection's document:

```
mongos> db.getCollection("version").find()
{ "_id" : 1, "version" : 3 }
```

Note: Like all databases in MongoDB, the config database contains a system.indexes (page 304) collection contains metadata for all indexes in the database for information on indexes, see *Indexes* (page 481).

CHAPTER 11

Frequently Asked Questions

11.1 FAQ: MongoDB Fundamentals

On this page

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 761) or post your question to the MongoDB User Mailing List[1].

11.1.1 What kind of database is MongoDB?

MongoDB is a *document*-oriented DBMS. Think of MySQL but with *JSON*-like objects comprising the data model, rather than RDBMS tables. Significantly, MongoDB supports neither joins nor transactions. However, it features secondary indexes, an expressive query language, and atomic writes on a per-document level.

Operationally, MongoDB features master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

Note: MongoDB uses *BSON*, a binary object format similar to, but more expressive than *JSON*.

[1] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

11.1.2 Do MongoDB databases have tables?

Instead of tables, a MongoDB database stores its data in *collections*, which are the rough equivalent of RDBMS tables. A collection holds one or more *documents*, which corresponds to a record or a row in a relational database table, and each document has one or more fields, which corresponds to a column in a relational database table.

Collections have important differences from RDBMS tables. Documents in a single collection may have a unique combination and set of fields. Documents need not have identical fields. You can add a field to some documents in a collection without adding that field to all documents in the collection.

See

SQL to MongoDB Mapping Chart (page 136)

11.1.3 Do MongoDB databases have schemas?

MongoDB uses dynamic schemas. You can create collections without defining the structure, i.e. the fields or the types of their values, of the documents in the collection. You can change the structure of documents simply by adding new fields or deleting existing ones. Documents in a collection need not have an identical set of fields.

In practice, it is common for the documents in a collection to have a largely homogeneous structure; however, this is not a requirement. MongoDB's flexible schemas mean that schema migration and augmentation are very easy in practice, and you will rarely, if ever, need to write scripts that perform "alter table" type operations, which simplifies and facilitates iterative software development with MongoDB.

See

SQL to MongoDB Mapping Chart (page 136)

11.1.4 What languages can I use to work with MongoDB?

MongoDB *client drivers* exist for all of the most popular programming languages, and many other ones. See the latest list of drivers[2] for details.

See also:

`http://docs.mongodb.org/manual/applications/drivers`.

11.1.5 Does MongoDB support SQL?

No.

However, MongoDB does support a rich, ad-hoc query language of its own.

See also:

`http://docs.mongodb.org/manual/reference/operator`

11.1.6 What are typical uses for MongoDB?

MongoDB has a general-purpose design, making it appropriate for a large number of use cases. Examples include content management systems, mobile applications, gaming, e-commerce, analytics, archiving, and logging.

[2] https://docs.mongodb.org/ecosystem/drivers

Do not use MongoDB for systems that require SQL, joins, and multi-object transactions.

11.1.7 Does MongoDB support transactions?

MongoDB does not support multi-document transactions. However, MongoDB does provide atomic operations on a single document.

For more details on MongoDB's isolation guarantees and behavior under concurrency, see *FAQ: Concurrency* (page 777).

11.1.8 Does MongoDB require a lot of RAM?

Not necessarily. It's certainly possible to run MongoDB on a machine with a small amount of free RAM.

MongoDB automatically uses all free memory on the machine as its cache. System resource monitors show that MongoDB uses a lot of memory, but its usage is dynamic. If another process suddenly needs half the server's RAM, MongoDB will yield cached memory to the other process.

Technically, the operating system's virtual memory subsystem manages MongoDB's memory. This means that MongoDB will use as much free memory as it can, swapping to disk as needed. Deployments with enough memory to fit the application's working data set in RAM will achieve the best performance.

See also:

FAQ: MongoDB Diagnostics (page 799) for answers to additional questions about MongoDB and Memory use.

11.1.9 How do I configure the cache size?

MongoDB has no configurable cache. MongoDB uses all *free* memory on the system automatically by way of memory-mapped files. Operating systems use the same approach with their file system caches.

11.1.10 Does MongoDB require a separate caching layer for application-level caching?

No. In MongoDB, a document's representation in the database is similar to its representation in application memory. This means the database already stores the usable form of data, making the data usable in both the persistent store and in the application cache. This eliminates the need for a separate caching layer in the application.

This differs from relational databases, where caching data is more expensive. Relational databases must transform data into object representations that applications can read and must store the transformed data in a separate cache: if these transformation from data to application objects require joins, this process increases the overhead related to using the database which increases the importance of the caching layer.

11.1.11 Does MongoDB handle caching?

Yes. MongoDB keeps all of the most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

MongoDB does not implement a query cache: MongoDB serves all queries directly from the indexes and/or data files.

11.1.12 Are writes written to disk immediately, or lazily?

Writes are physically written to the *journal* (page 309) 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 `commitIntervalMs` 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 `syncPeriodSecs`.

11.1.13 What language is MongoDB written in?

MongoDB is implemented in C++. *Drivers* and client libraries are typically written in their respective languages, although some drivers use C extensions for better performance.

11.1.14 What are the limitations of 32-bit versions of MongoDB?

MongoDB uses *memory-mapped files* (page 793). 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[3]"

Note: 32-bit builds disable *journaling* by default because journaling further limits the maximum amount of data that the database can store.

11.2 FAQ: MongoDB for Application Developers

[3] http://blog.mongodb.org/post/137788967/32-bit-limitations

On this page

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 761) or post your question to the MongoDB User Mailing List[4].

11.2.1 What is a namespace in MongoDB?

A "namespace" is the concatenation of the *database* name and the *collection* names [5] with a period character in between.

Collections are containers for documents that share one or more indexes. Databases are groups of collections stored on disk using a single set of data files. [6]

For an example `acme.users` namespace, `acme` is the database name and `users` is the collection name. Period characters **can** occur in collection names, so that `acme.user.history` is a valid namespace, with `acme` as the database name, and `user.history` as the collection name.

While data models like this appear to support nested collections, the collection namespace is flat, and there is no difference from the perspective of MongoDB between `acme`, `acme.users`, and `acme.records`.

11.2.2 How do you copy all objects from one collection to another?

In the `mongo` shell, you can use the following operation to duplicate the entire collection:

```
db.source.copyTo(newCollection)
```

[4] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
[5] Each index also has its own namespace.
[6] MongoDB database have a configurable limit on the `number of namespaces` in a database.

Warning: When using `db.collection.copyTo()` check field types to ensure that the operation does not remove type information from documents during the translation from *BSON* to *JSON*.
The `db.collection.copyTo()` method uses the `eval` command internally. As a result, the `db.collection.copyTo()` operation takes a global lock that blocks all other read and write operations until the `db.collection.copyTo()` completes.

Also consider the `cloneCollection` *command* that may provide some of this functionality.

11.2.3 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use `remove()`, the object will no longer exist in MongoDB's on-disk data storage.

11.2.4 When does MongoDB write updates to disk?

MongoDB flushes writes to disk on a regular interval. In the default configuration, MongoDB writes data to the main data files on disk every 60 seconds and commits the *journal* roughly every 100 milliseconds. These values are configurable with the `commitIntervalMs` and `syncPeriodSecs`.

These values represent the *maximum* amount of time between the completion of a write operation and the point when the write is durable in the journal, if enabled, and when MongoDB flushes data to the disk. In many cases MongoDB and the operating system flush data to disk more frequently, so that the above values represents a theoretical maximum.

However, by default, MongoDB uses a "lazy" strategy to write to disk. This is advantageous in situations where the database receives a thousand increments to an object within one second, MongoDB only needs to flush this data to disk once. In addition to the aforementioned configuration options, you can also use `fsync` and *Write Concern Reference* (page 135) to modify this strategy.

11.2.5 How do I do transactions and locking in MongoDB?

MongoDB does not have support for traditional locking or complex transactions with rollback. MongoDB aims to be lightweight, fast, and predictable in its performance. This is similar to the MySQL MyISAM autocommit model. By keeping transaction support extremely simple, MongoDB can provide greater performance especially for *partitioned* or *replicated* systems with a number of database server processes.

MongoDB *does* have support for atomic operations *within* a single document. Given the possibilities provided by nested documents, this feature provides support for a large number of use-cases.

See also:

The *Atomicity and Transactions* (page 86) page.

11.2.6 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new *aggregation framework* (page 439), with the `aggregate` command.

MongoDB also supports *map-reduce* with the `mapReduce` command, as well as basic aggregation with the `group`, `count`, and `distinct`. commands.

See also:

The *Aggregation* (page 435) page.

11.2.7 Why does MongoDB log so many "Connection Accepted" events?

If you see a very large number connection and re-connection messages in your MongoDB log, then clients are frequently connecting and disconnecting to the MongoDB server. This is normal behavior for applications that do not use request pooling, such as CGI. Consider using FastCGI, an Apache Module, or some other kind of persistent application server to decrease the connection overhead.

If these connections do not impact your performance you can use the run-time `quiet` option or the command-line option `--quiet` to suppress these messages from the log.

11.2.8 Does MongoDB run on Amazon EBS?

Yes.

MongoDB users of all sizes have had a great deal of success using MongoDB on the EC2 platform using EBS disks.

See also:

Amazon EC2[7]

11.2.9 Why are MongoDB's data files so large?

MongoDB aggressively preallocates data files to reserve space and avoid file system fragmentation. You can use the `storage.smallFiles` setting to modify the file preallocation strategy.

See also:

Why are the files in my data directory larger than the data in my database? (page 794)

11.2.10 How do I optimize storage use for small documents?

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the `_id` field explicitly.

  MongoDB clients automatically add an `_id` field to each document and generate a unique 12-byte *ObjectId* for the `_id` field. Furthermore, MongoDB always indexes the `_id` field. For smaller documents this may account for a significant amount of space.

  To optimize storage use, users can specify a value for the `_id` field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the `_id` field that would have occupied space in another portion of the document.

  You can store any value in the `_id` field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field's value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.

  MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of documents that resemble the following:

[7] https://docs.mongodb.org/ecosystem/platforms/amazon-ec2

```
{ last_name : "Smith", best_score: 3.9 }
```

If you shorten the field named `last_name` to `lname` and the field named `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 for larger documents and where document overhead is not of 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.

11.2.11 When should I use GridFS?

For documents in a MongoDB collection, you should always use *GridFS* for storing files larger than 16 MB.

In some situations, storing large files may be more efficient in a MongoDB database than on a system-level filesystem.

- If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed.
- When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities. When using *geographically distributed replica sets* (page 581) MongoDB can distribute files and their metadata automatically to a number of `mongod` instances and facilities.
- When you want to access information from portions of large files without having to load whole files into memory, you can use GridFS to recall sections of files without reading the entire file into memory.

Do not use GridFS if you need to update the content of the entire file atomically. As an alternative you can store multiple versions of each file and specify the current version of the file in the metadata. You can update the metadata field that indicates "latest" status in an atomic update after uploading the new version of the file, and later remove previous versions if needed.

Furthermore, if your files are all smaller the 16 MB `BSON Document Size` limit, consider storing the file manually within a single document. You may use the BinData data type to store the binary data. See your `drivers` documentation for details on using BinData.

For more information on GridFS, see *GridFS* (page 156).

11.2.12 How does MongoDB address SQL or Query injection?

BSON

As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks are not a problem. More details and some nuances are covered below.

MongoDB represents queries as *BSON* objects. Typically `client libraries` provide a convenient, injection free, process to build these objects. Consider the following C++ example:

```
BSONObj my_query = BSON( "name" << a_name );
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", my_query);
```

Here, `my_query` then will have a value such as `{ name : "Joe" }`. If `my_query` contained special characters, for example `,`, `:`, and `{`, the query simply wouldn't match any documents. For example, users cannot hijack a query and convert it to a delete.

JavaScript

Note: You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting `security.javascriptEnabled` in a configuration file.

All of the following MongoDB operations permit you to run arbitrary JavaScript expressions directly on the server:

- `$where`
- `db.eval()`
- `mapReduce`
- `group`

You must exercise care in these cases to prevent users from submitting malicious JavaScript.

Fortunately, you can express most queries in MongoDB without JavaScript and for queries that require JavaScript, you can mix JavaScript and non-JavaScript in a single query. Place all the user-supplied fields directly in a *BSON* field and pass JavaScript code to the `$where` field.

- If you need to pass user-supplied values in a `$where` clause, you may escape these values with the `CodeWScope` mechanism. When you set user-submitted values as variables in the scope document, you can avoid evaluating them on the database server.
- If you need to use `db.eval()` with user supplied values, you can either use a `CodeWScope` or you can supply extra arguments to your function. For instance:

  ```
  db.eval(function(userVal){...},
          user_value);
  ```

  This will ensure that your application sends `user_value` to the database server as data rather than code.

Dollar Sign Operator Escaping

Field names in MongoDB's query language have semantic meaning. The dollar sign (i.e `$`) is a reserved character used to represent `operators` (i.e. `$inc`.) Thus, you should ensure that your application's users cannot inject operators into their inputs.

In some cases, you may wish to build a BSON object with a user-provided key. In these situations, keys will need to substitute the reserved `$` and `.` characters. Any character is sufficient, but consider using the Unicode full width equivalents: `U+FF04` (i.e. "$") and `U+FF0E` (i.e. ".").

Consider the following example:

```
BSONObj my_object = BSON( a_key << a_name );
```

The user may have supplied a `$` value in the `a_key` value. At the same time, `my_object` might be `{ $where : "things" }`. Consider the following cases:

- **Insert**. Inserting this into the database does no harm. The insert process does not evaluate the object as a query.

  Note: MongoDB client drivers, if properly implemented, check for reserved characters in keys on inserts.

- **Update**. The `update()` operation permits `$` operators in the update argument but does not support the `$where` operator. Still, some users may be able to inject operators that can manipulate a single document only. Therefore your application should escape keys, as mentioned above, if reserved characters are possible.

- **Query** Generally this is not a problem for queries that resemble `{ x : user_obj }`: dollar signs are not top level and have no effect. Theoretically it may be possible for the user to build a query themselves. But checking the user-submitted content for `$` characters in key names may help protect against this kind of injection.

Driver-Specific Issues

See the "PHP MongoDB Driver Security Notes[8]" page in the PHP driver documentation for more information

11.2.13 How does MongoDB provide concurrency?

MongoDB implements a readers-writer lock. This means that at any one time, only one client may be writing or any number of clients may be reading, but that reading and writing cannot occur simultaneously.

In standalone and *replica sets* the lock's scope applies to a single `mongod` instance or *primary* instance. In a sharded cluster, locks apply to each individual shard, not to the whole cluster.

For more information, see *FAQ: Concurrency* (page 777).

11.2.14 What is the compare order for BSON types?

MongoDB permits documents within a single collection to have fields with different *BSON* types. For instance, the following documents may exist within a single collection.

```
{ x: "string" }
{ x: 42 }
```

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

The comparison treats a non-existent field as it would an empty BSON Object. As such, a sort on the `a` field in documents `{ }` and `{ a: null }` would treat the documents as equivalent in sort order.

[8] http://us.php.net/manual/en/mongo.security.php

With arrays, a less-than comparison or an ascending sort compares the smallest element of arrays, and a greater-than comparison or a descending sort compares the largest element of the arrays. As such, when comparing a field whose value is a single-element array (e.g. `[ 1 ]`) with non-array fields (e.g. `2`), the comparison is between `1` and `2`. A comparison of an empty array (e.g. `[ ]`) treats the empty array as less than `null` or a missing field.

MongoDB sorts `BinData` in the following order:

1. First, the length or size of the data.
2. Then, by the BSON one-byte subtype.
3. Finally, by the data, performing a byte-by-byte comparison.

Consider the following `mongo` example:

```
db.test.insert( {x : 3 } );
db.test.insert( {x : 2.9 } );
db.test.insert( {x : new Date() } );
db.test.insert( {x : true } );

db.test.find().sort({x:1});
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
```

The `$type` operator provides access to *BSON type* comparison in the MongoDB query syntax. See the documentation on *BSON types* and the `$type` operator for additional information.

Warning: Data models that associate a field name with different data types within a collection are *strongly* discouraged.
Without internal consistency complicates application code, and can lead to unnecessary complexity for application developers.

See also:

- The *Tailable Cursors* (page 128) page for an example of a C++ use of `MinKey`.

11.2.15 When multiplying values of mixed types, what type conversion rules apply?

The `$mul` multiplies the numeric value of a field by a number. For multiplication with values of mixed numeric types (32-bit integer, 64-bit integer, float), the following type conversion rules apply:

	32-bit Integer	64-bit Integer	Float
32-bit Integer	32-bit or 64-bit Integer	64-bit Integer	Float
64-bit Integer	64-bit Integer	64-bit Integer	Float
Float	Float	Float	Float

Note:

- If the product of two 32-bit integers exceeds the maximum value for a 32-bit integer, the result is a 64-bit integer.
- Integer operations of any type that exceed the maximum value for a 64-bit integer produce an error.

11.2.16 How do I query for fields that have null values?

Different query operators treat `null` values differently.

Consider the collection `test` with the following documents:

```
{ _id: 1, cancelDate: null }
{ _id: 2 }
```

Comparison with Null

The `{ cancelDate : null }` query matches documents that either contain the `cancelDate` field whose value is `null` *or* that do not contain the `cancelDate` field. If the queried index is *sparse* (page 507), however, then the query will only match `null` values, not missing fields.

Changed in version 2.6: If using the sparse index results in an incomplete result, MongoDB will not use the index unless a `hint()` explicitly specifies the index. See *Sparse Indexes* (page 507) for more information.

Given the following query:

```
db.test.find( { cancelDate: null } )
```

The query returns both documents:

```
{ "_id" : 1, "cancelDate" : null }
{ "_id" : 2 }
```

Type Check

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 }
```

Existence Check

The `{ cancelDate : { $exists: false } }` query matches documents that do not contain the `cancelDate` field:

```
db.test.find( { cancelDate : { $exists: false } } )
```

The query returns only the document that does *not* contain the `cancelDate` field:

```
{ "_id" : 2 }
```

See also:

The reference documentation for the `$type` and `$exists` operators.

11.2.17 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 length of the collection namespace, which includes the database name, the dot (`.`) separator, and the collection name (i.e. `<database>.<collection>`), is 120 bytes.

  However, for maximum flexibility, collections should have names less than 80 characters.

If your collection name includes special characters, such as the underscore character, then to access the collection use the `db.getCollection()` method or a similar method for your driver[9].

Example

To create a collection `_foo` and insert the `{ a : 1 }` document, use the following operation:

```
db.getCollection("_foo").insert( { a : 1 } )
```

To perform a query, use the `find()` method, in as the following:

```
db.getCollection("_foo").find()
```

11.2.18 How do I isolate cursors from intervening write operations?

MongoDB cursors can return the same document more than once in some situations. [10] You can use the `snapshot()` method on a cursor to isolate the operation for a very specific case.

`snapshot()` traverses the index on the `_id` field and guarantees that the query will return each document (with respect to the value of the `_id` field) no more than once. [11]

The `snapshot()` does not guarantee that the data returned by the query will reflect a single moment in time *nor* does it provide isolation from insert or delete operations.

Warning:
- You **cannot** use `snapshot()` with *sharded collections*.
- You **cannot** use `snapshot()` with `sort()` or `hint()` cursor methods.

As an alternative, if your collection has a field or fields that are never modified, you can use a *unique* index on this field or these fields to achieve a similar result as the `snapshot()`. Query with `hint()` to explicitly force the query to use that index.

11.2.19 When should I embed documents within other documents?

When *modeling data in MongoDB* (page 151), 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.

[9] https://api.mongodb.org/

[10] As a cursor returns documents other operations may interleave with the query: if some of these operations are *updates* (page 77) that cause the document to move (in the case of a table scan, caused by document growth) or that change the indexed field on the index used by the query; then the cursor will return the same document more than once.

[11] MongoDB does not permit changes to the value of the `_id` field; it is not possible for a cursor that transverses this index to pass the same document more than once.

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 embedded documents. Keep in mind that if you often only need to retrieve a subset of the documents within the group, then "rolling-up" the documents may not provide better performance.

"Rolling up" these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses.

Additionally, "rolling up" documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields *and* there would be fewer associated key entries in the corresponding index. See *Index Concepts* (page 485) for more information on indexes.

11.2.20 Where can I learn more about data modeling in MongoDB?

Begin by reading the documents in the *Data Models* (page 149) section. These documents contain a high level introduction to data modeling considerations in addition to practical examples of data models targeted at particular issues.

Additionally, consider the following external resources that provide additional examples:

- Schema Design by Example[12]
- Dynamic Schema Blog Post[13]
- MongoDB Data Modeling and Rails[14]
- Ruby Example of Materialized Paths[15]
- Sean Cribs Blog Post[16] which was the source for much of the *data-modeling-trees* content.

11.2.21 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*.

You should not have to pad manually because MongoDB adds *padding automatically* (page 95) and can adaptively adjust the amount of padding added to documents to prevent document relocations following updates. You can change the default `paddingFactor` calculation by using the `collMod` command with the `usePowerOf2Sizes` flag. The `usePowerOf2Sizes` flag ensures that MongoDB allocates document space in sizes that are powers of 2, which helps ensure that MongoDB can efficiently reuse free space created by document deletion or relocation.

However, *if you must* pad a document manually, you can add a temporary field to the document and then `$unset` the field, as in the following example.

Warning: Do not manually pad documents in a capped collection. Applying manual padding to a document in a capped collection can break replication. Also, the padding is not preserved if you re-sync the MongoDB instance.

[12] http://www.mongodb.com/presentations/mongodb-melbourne-2012/schema-design-example
[13] http://dmerr.tumblr.com/post/6633338010/schemaless
[14] https://docs.mongodb.org/ecosystem/tutorial/model-data-for-ruby-on-rails/
[15] http://github.com/banker/newsmonger/blob/master/app/models/comment.rb
[16] http://seancribbs.com/tech/2009/09/28/modeling-a-tree-in-a-document-database

```
var myTempPadding = [ "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"];

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:

Record Allocation Strategies (page 95)

11.3 FAQ: The mongo Shell

On this page

11.3.1 How can I enter multi-line operations in the mongo shell?

If you end a line with an open parenthesis (`'('`), an open brace (`'{'`), or an open bracket (`'['`), then the subsequent lines start with ellipsis (`"..."`) until you enter the corresponding closing parenthesis (`')'`), the closing brace (`'}'`) or the closing bracket (`']'`). The mongo shell waits for the closing parenthesis, closing brace, or the closing bracket before evaluating the code, as in the following example:

```
> if ( x > 0 ) {
... count++;
... print (x);
... }
```

You can exit the line continuation mode if you enter two blank lines, as in the following example:

```
> if (x > 0
...
...
>
```

11.3.2 How can I access different databases temporarily?

You can use `db.getSiblingDB()` method to access another database without switching databases, as in the following example which first switches to the `test` database and then accesses the `sampleDB` database from the `test`

database:

```
use test

db.getSiblingDB('sampleDB').getCollectionNames();
```

11.3.3 Does the mongo shell support tab completion and other keyboard shortcuts?

The mongo shell supports keyboard shortcuts. For example,

- Use the up/down arrow keys to scroll through command history. See *.dbshell* documentation for more information on the .dbshell file.
- Use <Tab> to autocomplete or to list the completion possibilities, as in the following example which uses <Tab> to complete the method name starting with the letter 'c':

  ```
  db.myCollection.c<Tab>
  ```

  Because there are many collection methods starting with the letter 'c', the <Tab> will list the various methods that start with 'c'.

For a full list of the shortcuts, see *Shell Keyboard Shortcuts*

11.3.4 How can I customize the mongo shell prompt?

New in version 1.9.

You can change the mongo shell prompt by setting the prompt variable. This makes it possible to display additional information in the prompt.

Set prompt to any string or arbitrary JavaScript code that returns a string, consider the following examples:

- Set the shell prompt to display the hostname and the database issued:

  ```
  var host = db.serverStatus().host;
  var prompt = function() { return db+"@"+host+"> "; }
  ```

  The mongo shell prompt should now reflect the new prompt:

  ```
  test@my-machine.local>
  ```

- Set the shell prompt to display the database statistics:

  ```
  var prompt = function() {
                  return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > "
               }
  ```

  The mongo shell prompt should now reflect the new prompt:

  ```
  Uptime:1052 Documents:25024787 >
  ```

You can add the logic for the prompt in the *.mongorc.js* file to set the prompt each time you start up the mongo shell.

11.3.5 Can I edit long shell operations with an external text editor?

You can use your own editor in the mongo shell by setting the EDITOR environment variable before starting the mongo shell. Once in the mongo shell, you can edit with the specified editor by typing edit <variable> or edit <function>, as in the following example:

1. Set the `EDITOR` variable from the command line prompt:

   ```
   EDITOR=vim
   ```

2. Start the `mongo` shell:

   ```
   mongo
   ```

3. Define a function `myFunction`:

   ```
   function myFunction () { }
   ```

4. Edit the function using your editor:

   ```
   edit myFunction
   ```

   The command should open the `vim` edit session. Remember to save your changes.

5. Type `myFunction` to see the function definition:

   ```
   myFunction
   ```

   The result should be the changes from your saved edit:

   ```
   function myFunction() {
       print("This was edited");
   }
   ```

11.4 FAQ: Concurrency

On this page

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 the same 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[17]

[17] http://www.mongodb.com/presentations/concurrency-internals-mongodb-2-2

11.4.1 What type of locking does MongoDB use?

MongoDB uses a readers-writer [18] 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 write locks have preference over reads. When both a read and write are waiting for a lock, MongoDB grants the lock to the write.

11.4.2 How granular are locks in MongoDB?

Changed in version 2.2.

Beginning with version 2.2, MongoDB implements locks on a per-database basis for most read and write operations. Some global operations, typically short lived operations involving multiple databases, still require a global "instance" wide lock. Before 2.2, there is only one "global" lock per `mongod` instance.

For example, if you have six databases and one takes a database-level write lock, the other five are still available for read and write. A global lock makes all six databases unavailable during the operation.

11.4.3 How do I see the status of locks on my `mongod` instances?

For reporting on lock utilization information on locks, use any of the following methods:

- `db.serverStatus()`,
- `db.currentOp()`,
- `mongotop`,
- `mongostat`, and/or
- the MongoDB Cloud Manager[19] or Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[20]

Specifically, the `locks` document in the `output of serverStatus`, or the `locks` field in the `current operation reporting` provides insight into the type of locks and amount of lock contention in your `mongod` instance.

To terminate an operation, use `db.killOp()`.

11.4.4 Does a read or write operation ever yield the lock?

In some situations, read and write operations can yield their locks.

Long running read and write operations, such as queries, updates, and deletes, yield under many conditions. MongoDB uses an adaptive algorithms to allow operations to yield locks based on predicted disk access patterns (i.e. page faults.)

MongoDB operations can also yield locks between individual document modification in write operations that affect multiple documents like `update()` with the `multi` parameter.

[18] You may be familiar with a "readers-writer" lock as "multi-reader" or "shared exclusive" lock. See the Wikipedia page on Readers-Writer Locks (http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock) for more information.

[19] https://cloud.mongodb.com/?jmp=docs

[20] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

MongoDB uses heuristics based on its access pattern to predict whether data is likely in physical memory before performing a read. If MongoDB *predicts* that the data is not in physical memory an operation will yield its lock while MongoDB loads the data to memory. Once data is available in memory, the operation will reacquire the lock to complete the operation.

Changed in version 2.6: MongoDB does not yield locks when scanning an index even if it predicts that the index is not in memory.

11.4.5 Which operations lock the database?

Changed in version 2.2.

The following table lists common database operations and the types of locks they use.

Operation	Lock Type
Issue a query	Read lock
Get more data from a *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()`	Write lock. The `db.eval()` method takes a global write lock while evaluating the JavaScript function. To avoid taking this global write lock, you can use the `eval` command with `nolock: true`.
`eval`	Write lock. By default, `eval` command takes a global write lock while evaluating the JavaScript function. If used with `nolock: true`, the `eval` command does *not* take a global write lock while evaluating the JavaScript function. However, the logic within the JavaScript function may take write locks for write operations.
`aggregate()`	Read lock

11.4.6 Which administrative commands lock the database?

Certain administrative commands can exclusively lock the database for extended periods of time. In some deployments, for large databases, you may consider taking the `mongod` instance offline so that clients are not affected. For example, if a `mongod` is part of a *replica set*, take the `mongod` offline and let other members of the set service load while maintenance is in progress.

The following administrative operations require an exclusive (i.e. write) lock on the database for extended periods:

- `db.collection.ensureIndex()`, when issued *without* setting `background` to `true`,
- `reIndex`,
- `compact`,
- `db.repairDatabase()`,
- `db.createCollection()`, when creating a very large (i.e. many gigabytes) capped collection,
- `db.collection.validate()`, and
- `db.copyDatabase()`. This operation may lock all databases. See *Does a MongoDB operation ever lock more than one database?* (page 780).

The following administrative commands lock the database but only hold the lock for a very short time:

- `db.collection.dropIndex()`,
- `db.getLastError()`,
- `db.isMaster()`,
- `rs.status()` (i.e. `replSetGetStatus`),
- `db.serverStatus()`,
- `db.auth()`, and
- `db.addUser()`.

11.4.7 Does a MongoDB operation ever lock more than one database?

The following MongoDB operations lock multiple databases:

- `db.copyDatabase()` must lock the entire `mongod` instance at once.
- `db.repairDatabase()` obtains a global write lock and will block other operations until it finishes.
- *Journaling*, which is an internal operation, locks all databases for short intervals. All databases share a single journal.
- *User authentication* (page 316) requires a read lock on the `admin` database for deployments using *2.6 user credentials* (page 415). For deployments using the 2.4 schema for user credentials, authentication locks the `admin` database as well as the database the user is accessing.
- All writes to a replica set's *primary* lock both the database receiving the writes and then the `local` database for a short time. The lock for the `local` database allows the `mongod` to write to the primary's *oplog* and accounts for a small portion of the total time of the operation.

11.4.8 How does sharding affect concurrency?

Sharding improves concurrency by distributing collections over multiple `mongod` instances, allowing shard servers (i.e. `mongos` processes) to perform any number of operations concurrently to the various downstream `mongod` instances.

Each `mongod` instance is independent of the others in the shard cluster and uses the MongoDB *readers-writer lock* (page 778). The operations on one `mongod` instance do not block the operations on any others.

11.4.9 How does concurrency affect a replica set primary?

In *replication*, when MongoDB writes to a collection on the *primary*, MongoDB also writes to the primary's *oplog*, which is a special collection in the `local` database. Therefore, MongoDB must lock both the collection's database and the `local` database. The `mongod` must lock both databases at the same time to keep the database consistent and ensure that write operations, even with replication, are "all-or-nothing" operations.

11.4.10 How does concurrency affect secondaries?

In *replication*, MongoDB does not apply writes serially to *secondaries*. Secondaries collect oplog entries in batches and then apply those batches in parallel. Secondaries do not allow reads while applying the write operations, and apply write operations in the order that they appear in the oplog.

MongoDB can apply several writes in parallel on replica set secondaries, in two phases:

1. During the first *prefer* phase, under a read lock, the `mongod` ensures that all documents affected by the operations are in memory. During this phase, other clients may execute queries against this member.
2. A thread pool using write locks applies all write operations in the batch as part of a coordinated write phase.

11.4.11 What kind of concurrency does MongoDB provide for JavaScript operations?

Changed in version 2.4: The V8 JavaScript engine added in 2.4 allows multiple JavaScript operations to run at the same time. Prior to 2.4, a single `mongod` could only run a *single* JavaScript operation at once.

11.4.12 Does MongoDB support transactions?

MongoDB does not support multi-document transactions.

However, MongoDB does provide atomic operations on a single document. Often these document-level atomic operations are sufficient to solve problems that would require ACID transactions in a relational database.

For example, in MongoDB, you can embed related data in nested arrays or nested documents within a single document and update the entire document in a single atomic operation. Relational databases might represent the same kind of data with multiple tables and rows, which would require transaction support to update the data atomically.

See also:

Atomicity and Transactions (page 86)

11.4.13 What isolation guarantees does MongoDB provide?

MongoDB provides the following guarantees in the presence of concurrent read and write operations.

1. Read and write operations are atomic with respect to a single document and will always leave the document in a consistent state. This means that readers will never see a partially updated document, and indices will always be consistent with the contents of the collection. Furthermore, a set of read and write operations to a single document are serializable.
2. Correctness with respect to query predicates, e.g. `db.collection.find()` will only return documents that match and `db.collection.update()` will only write to matching documents.
3. Correctness with respect to sort. For read operations that request a sort order (e.g. `db.collection.find()` or `db.collection.aggregate()`), the sort order will not be violated due to concurrent writes.

Although MongoDB provides these strong guarantees for single-document operations, read and write operations may access an arbitrary number of documents during execution. Multi-document operations do *not* occur transactionally and are not isolated from concurrent writes. This means that the following behaviors are expected under the normal operation of the system:

1. Non-point-in-time read operations. Suppose a read operation begins at time t_1 and starts reading documents. A write operation then commits an update to a document at some later time t_2. The reader may see the updated version of the document, and therefore does not see a point-in-time snapshot of the data.
2. Non-serializable operations. Suppose a read operation reads a document d_1 at time t_1 and a write operation updates d_1 at some later time t_3. This introduces a read-write dependency such that, if the operations were to be serialized, the read operation must precede the write operation. But also suppose that the write operation updates document d_2 at time t_2 and the read operation subsequently reads d_2 at some later time t_4. This introduces a write-read dependency which would instead require the read operation to come *after* the write operation in a serializable schedule. There is a dependency cycle which makes serializability impossible.

3. Dropped results. Reads may miss matching documents that are updated or deleted during the course of the read operation. However, data that has not been modified during the operation will always be visible.

See also:

Atomicity and Transactions (page 86)

11.4.14 Can reads see changes that have not been committed to disk?

Yes, readers can see the results of writes before they are made durable, regardless of write concern level or journaling configuration. As a result, applications may observe the following behaviors:

1. MongoDB will allow a concurrent reader to see the result of the write operation before the write is acknowledged to the client application. For details on when writes are acknowledged for different write concern levels, see *Write Concern* (page 82).
2. Reads can see data which may subsequently be rolled back in rare cases such as replica set failover or power loss. It does *not* mean that read operations can see documents in a partially written or otherwise inconsistent state.

Other systems refer to these semantics as *read uncommitted*.

11.5 FAQ: Sharding with MongoDB

On this page

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 761) or post your question to the MongoDB User Mailing List[21].

11.5.1 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.

11.5.2 How does sharding work with replication?

To use replication with sharding, deploy each *shard* as a *replica set*.

[21] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

11.5.3 Can I change the shard key after sharding a collection?

No.

There is no automatic support in MongoDB for changing a shard key after sharding a collection. This reality underscores the importance of choosing a good *shard key* (page 687). 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 738) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See `shardCollection`, `sh.shardCollection()`, the *Shard Key* (page 687), *Deploy a Sharded Cluster* (page 705), and SERVER-4000[22] for more information.

11.5.4 What happens to unsharded collections in sharded databases?

In the current implementation, all databases in a *sharded cluster* have a "primary *shard*." All unsharded collection within that database will reside on the same shard.

11.5.5 How does MongoDB distribute data across shards?

Sharding must be specifically enabled on a collection. After enabling sharding on the collection, MongoDB will assign various ranges of collection data to the different shards in the cluster. The cluster automatically corrects imbalances between shards by migrating ranges of data from one shard to another.

11.5.6 What happens if a client updates a document in a chunk during a migration?

The `mongos` routes the operation to the "old" shard, where it will succeed immediately. Then the *shard* `mongod` instances will replicate the modification to the "new" shard before the *sharded cluster* updates that chunk's "ownership," which effectively finalizes the migration process.

11.5.7 What happens to queries if a shard is inaccessible or slow?

If a *shard* is inaccessible or unavailable, queries will return with an error.

However, a client may set the `partial` query bit, which will then return results from all available shards, regardless of whether a given shard is unavailable.

If a shard is responding slowly, `mongos` will merely wait for the shard to return results.

11.5.8 How does MongoDB distribute queries among shards?

Changed in version 2.0.

[22] https://jira.mongodb.org/browse/SERVER-4000

The exact method for distributing queries to *shards* in a *cluster* depends on the nature of the query and the configuration of the sharded cluster. Consider a sharded collection, using the *shard key* `user_id`, that has `last_login` and `email` attributes:

- For a query that selects one or more values for the `user_id` key:

  `mongos` determines which shard or shards contains the relevant data, based on the cluster metadata, and directs a query to the required shard or shards, and returns those results to the client.

- For a query that selects `user_id` and also performs a sort:

  `mongos` can make a straightforward translation of this operation into a number of queries against the relevant shards, ordered by `user_id`. When the sorted queries return from all shards, the `mongos` merges the sorted results and returns the complete result to the client.

- For queries that select on `last_login`:

  These queries must run on all shards: `mongos` must parallelize the query over the shards and perform a merge-sort on the `email` of the documents found.

11.5.9 How does MongoDB sort queries in sharded environments?

If you call the `cursor.sort()` method on a query in a sharded environment, the `mongod` for each shard will sort its results, and the `mongos` merges each shard's results before returning them to the client.

11.5.10 How does MongoDB ensure unique `_id` field values when using a shard key *other* than `_id`?

If you do not use `_id` as the shard key, then your application/client layer must be responsible for keeping the `_id` field unique. It is problematic for collections to have duplicate `_id` values.

If you're not sharding your collection by the `_id` field, then you should be sure to store a globally unique identifier in that field. The default *BSON ObjectId* (page 184) works well in this case.

11.5.11 I've enabled sharding and added a second shard, but all the data is still on one server. Why?

First, ensure that you've declared a *shard key* for your collection. Until you have configured the shard key, MongoDB will not create *chunks*, and *sharding* will not occur.

Next, keep in mind that the default chunk size is 64 MB. As a result, in most situations, the collection needs to have at least 64 MB of data before a migration will occur.

Additionally, the system which balances chunks among the servers attempts to avoid superfluous migrations. Depending on the number of shards, your shard key, and the amount of data, systems often require at least 10 chunks of data to trigger migrations.

You can run `db.printShardingStatus()` to see all the chunks present in your cluster.

11.5.12 Is it safe to remove old files in the `moveChunk` directory?

Yes. `mongod` creates these files as backups during normal *shard* balancing operations. If some error occurs during a *migration* (page 700), these files may be helpful in recovering documents affected during the migration.

Once the migration has completed successfully and there is no need to recover documents from these files, you may safely delete these files. Or, if you have an existing backup of the database that you can use for recovery, you may also delete these files after migration.

To determine if all migrations are complete, run `sh.isBalancerRunning()` while connected to a `mongos` instance.

11.5.13 How does `mongos` use connections?

Each client maintains a connection to a `mongos` instance. Each `mongos` instance maintains a pool of connections to the members of a replica set supporting the sharded cluster. Clients use connections between `mongos` and `mongod` instances one at a time. Requests are not multiplexed or pipelined. When client requests complete, the `mongos` returns the connection to the pool.

See the *System Resource Utilization* (page 300) section of the *UNIX ulimit Settings* (page 300) document.

11.5.14 Why does `mongos` hold connections open?

`mongos` uses a set of connection pools to communicate with each *shard*. These pools do not shrink when the number of clients decreases.

This can lead to an unused `mongos` with a large number of open connections. If the `mongos` is no longer in use, it is safe to restart the process to close existing connections.

11.5.15 Where does MongoDB report on connections used by `mongos`?

Connect to the `mongos` with the `mongo` shell, and run the following command:

```
db._adminCommand("connPoolStats");
```

11.5.16 What does `writebacklisten` in the log mean?

The writeback listener is a process that opens a long poll to relay writes back from a `mongod` or `mongos` after migrations to make sure they have not gone to the wrong server. The writeback listener sends writes back to the correct server if necessary.

These messages are a key part of the sharding infrastructure and should not cause concern.

11.5.17 How should administrators deal with failed migrations?

Failed migrations require no administrative intervention. Chunk migrations always preserve a consistent state. If a migration fails to complete for some reason, the *cluster* retries the operation. When the migration completes successfully, the data resides only on the new shard.

11.5.18 What is the process for moving, renaming, or changing the number of config servers?

See *Sharded Cluster Tutorials* (page 704) for information on migrating and replacing config servers.

11.5.19 When do the mongos servers detect config server changes?

mongos instances maintain a cache of the *config database* that holds the metadata for the *sharded cluster*. This metadata includes the mapping of *chunks* to *shards*.

mongos updates its cache lazily by issuing a request to a shard and discovering that its metadata is out of date. There is no way to control this behavior from the client, but you can run the flushRouterConfig command against any mongos to force it to refresh its cache.

11.5.20 Is it possible to quickly update mongos servers after updating a replica set configuration?

The mongos instances will detect these changes without intervention over time. However, if you want to force the mongos to reload its configuration, run the flushRouterConfig command against to each mongos directly.

11.5.21 What does the maxConns setting on mongos do?

The maxIncomingConnections option limits the number of connections accepted by mongos.

If your client driver or application creates a large number of connections but allows them to time out rather than closing them explicitly, then it might make sense to limit the number of connections at the mongos layer.

Set maxIncomingConnections to a value slightly higher than the maximum number of connections that the client creates, or the maximum size of the connection pool. This setting prevents the mongos from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

11.5.22 How do indexes impact queries in sharded systems?

If the query does not include the *shard key*, the mongos must send the query to all shards as a "scatter/gather" operation. Each shard will, in turn, use *either* the shard key index or another more efficient index to fulfill the query.

If the query includes multiple sub-expressions that reference the fields indexed by the shard key *and* the secondary index, the mongos can route the queries to a specific shard and the shard will use the index that will allow it to fulfill most efficiently. See this presentation[23] for more information.

11.5.23 Can shard keys be randomly generated?

Shard keys can be random. Random keys ensure optimal distribution of data across the cluster.

Sharded clusters, attempt to route queries to *specific* shards when queries include the shard key as a parameter, because these directed queries are more efficient. In many cases, random keys can make it difficult to direct queries to specific shards.

11.5.24 Can shard keys have a non-uniform distribution of values?

Yes. There is no requirement that documents be evenly distributed by the shard key.

However, documents that have the same 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.

[23] http://www.slideshare.net/mongodb/how-queries-work-with-sharding

11.5.25 Can you shard on the `_id` field?

You can use any field for the shard key. The `_id` field is a common shard key.

Be aware that `ObjectId()` values, which are the default value of the `_id` field, increment as a timestamp. As a result, when used as a shard key, all new documents inserted into the collection will initially belong to the same chunk on a single shard. Although the system will eventually divide this chunk and migrate its contents to distribute data more evenly, at any moment the cluster can only direct insert operations at a single shard. This can limit the throughput of inserts. If most of your write operations are updates, this limitation should not impact your performance. However, if you have a high insert volume, this may be a limitation.

To address this issue, MongoDB 2.4 provides *hashed shard keys* (page 689).

11.5.26 What do `moveChunk commit failed` errors mean?

At the end of a *chunk migration* (page 700), the *shard* must connect to the *config database* to update the chunk's record in the cluster metadata. If the *shard* fails to connect to the *config database*, MongoDB reports the following error:

```
ERROR: moveChunk commit failed: version is at <n>|<nn> instead of
<N>|<NN>" and "ERROR: TERMINATING"
```

When this happens, the *primary* member of the shard's replica set then terminates to protect data consistency. If a *secondary* member can access the config database, data on the shard becomes accessible again after an election.

The user will need to resolve the chunk migration failure independently. If you encounter this issue, contact the MongoDB User Group[24] or MongoDB Support[25] to address this issue.

11.5.27 How does draining a shard affect the balancing of uneven chunk distribution?

The sharded cluster balancing process controls both migrating chunks from decommissioned shards (i.e. draining) and normal cluster balancing activities. Consider the following behaviors for different versions of MongoDB in situations where you remove a shard in a cluster with an uneven chunk distribution:

- After MongoDB 2.2, the balancer first removes the chunks from the draining shard and then balances the remaining uneven chunk distribution.
- Before MongoDB 2.2, the balancer handles the uneven chunk distribution and *then* removes the chunks from the draining shard.

11.6 FAQ: Replication and Replica Sets

[24] http://groups.google.com/group/mongodb-user
[25] https://www.mongodb.org/about/support

On this page

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 761) or post your question to the MongoDB User Mailing List[26].

11.6.1 What kinds of replication does MongoDB support?

MongoDB supports master-slave replication and a variation on master-slave replication known as replica sets. Replica sets are the recommended replication topology.

11.6.2 What do the terms "primary" and "master" mean?

Primary and *master* nodes are the nodes that can accept writes. MongoDB's replication is "single-master:" only one node can accept write operations at a time.

In a replica set, if the current "primary" node fails or becomes inaccessible, the other members can autonomously *elect* one of the other members of the set to be the new "primary".

By default, clients send all reads to the primary; however, *read preference* is configurable at the client level on a per-connection basis, which makes it possible to send reads to secondary nodes instead.

11.6.3 What do the terms "secondary" and "slave" mean?

Secondary and *slave* nodes are read-only nodes that replicate from the *primary*.

Replication operates by way of an *oplog*, from which secondary/slave members apply new operations to themselves. This replication process is asynchronous, so secondary/slave nodes may not always reflect the latest writes to the primary. But usually, the gap between the primary and secondary nodes is just few milliseconds on a local network connection.

11.6.4 How long does replica set failover take?

It varies, but a replica set will select a new primary within a minute.

[26] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

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.

11.6.5 Does replication work over the Internet and WAN connections?

Yes.

For example, a deployment may maintain a *primary* and *secondary* in an East-coast data center along with a *secondary* member for disaster recovery in a West-coast data center.

See also:

Deploy a Geographically Redundant Replica Set (page 612)

11.6.6 Can MongoDB replicate over a "noisy" connection?

Yes, but not without connection failures and the obvious latency.

Members of the set will attempt to reconnect to the other members of the set in response to networking flaps. This does not require administrator intervention. However, if the network connections among the nodes in the replica set are very slow, it might not be possible for the members of the node to keep up with the replication.

If the TCP connection between the secondaries and the *primary* instance breaks, a *replica set* will automatically elect one of the *secondary* members of the set as primary.

11.6.7 What is the preferred replication method: master/slave or replica sets?

New in version 1.8.

Replica sets are the preferred *replication* mechanism in MongoDB. However, if your deployment requires more than 12 nodes, you must use master/slave replication.

11.6.8 What is the preferred replication method: replica sets or replica pairs?

Deprecated since version 1.6.

Replica sets replaced *replica pairs* in version 1.6. *Replica sets* are the preferred *replication* mechanism in MongoDB.

11.6.9 Why use journaling if replication already provides data redundancy?

Journaling facilitates faster crash recovery. Prior to journaling, crashes often required `database repairs` or full data resync. Both were slow, and the first was unreliable.

Journaling is particularly useful for protection against power failures, especially if your replica set resides in a single data center or power circuit.

When a *replica set* runs with journaling, `mongod` instances can safely restart without any administrator intervention.

Note: Journaling requires some resource overhead for write operations. Journaling has no effect on read performance,

however.

Journaling is enabled by default on all 64-bit builds of MongoDB v2.0 and greater.

11.6.10 Are write operations durable if write concern does not acknowledge writes?

Yes.

However, if you want confirmation that a given write has arrived at the server, use *write concern* (page 82).

After the *default write concern change* (page 907), the default write concern acknowledges all write operations, and unacknowledged writes must be explicitly configured. See the `http://docs.mongodb.org/manual/applications/drivers` documentation for your driver for more information.

Changed in version 2.6: The `mongo` shell now defaults to use *safe writes* (page 82). See *Write Method Acknowledgements* (page 838) for more information.

A new protocol for *write operations* (page 832) integrates write concerns with the write operations. Previous versions issued a `getLastError` command after a write to specify a write concern.

11.6.11 How many arbiters do replica sets need?

Some configurations do not require any *arbiter* instances. Arbiters vote in *elections* for *primary* but do not replicate the data like *secondary* members.

Replica sets require a majority of the remaining nodes present to elect a primary. Arbiters allow you to construct this majority without the overhead of adding replicating nodes to the system.

There are many possible replica set *architectures* (page 575).

A replica set with an odd number of voting nodes does not need an arbiter.

A common configuration consists of two replicating nodes that include a *primary* and a *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 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 575)

11.6.12 What information do arbiters exchange with the rest of the replica set?

Arbiters never receive the contents of a collection but do exchange the following data with the rest of the replica set:

- Credentials used to authenticate the arbiter with the replica set. All MongoDB processes within a replica set use keyfiles. These exchanges are encrypted.
- Replica set configuration data and voting data. This information is not encrypted. Only credential exchanges are encrypted.

If your MongoDB deployment uses TLS/SSL, then all communications between arbiters and the other members of the replica set are secure. See the documentation for *Configure mongod and mongos for TLS/SSL* (page 338) for more information. Run all arbiters on secure networks, as with all MongoDB components.

See

The overview of *Arbiter Members of Replica Sets* (page **??**).

11.6.13 Which members of a replica set vote in elections?

All members of a replica set, unless the value of `votes` (page 663) is equal to `0`, vote in elections. This includes all *delayed* (page 573), *hidden* (page 572) and *secondary-only* (page 570) members, as well as the *arbiters* (page **??**).

Additionally, the `state` of the voting members also determine whether the member can vote. Only voting members in the following states are eligible to vote:

- `PRIMARY`
- `SECONDARY`
- `RECOVERING`
- `ARBITER`
- `ROLLBACK`

See also:

Replica Set Elections (page 583)

11.6.14 Do hidden members vote in replica set elections?

Hidden members (page 572) 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 663) configuration to `0`.

See also:

Replica Set Elections (page 583)

11.6.15 Is it normal for replica set members to use different amounts of disk space?

Yes.

Factors including: different oplog sizes, different levels of storage fragmentation, and MongoDB's data file pre-allocation can lead to some variation in storage utilization between nodes. Storage use disparities will be most pronounced when you add members at different times.

11.7 FAQ: MongoDB Storage

On this page

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 761) or post your question to the MongoDB User Mailing List[27].

11.7.1 What are memory mapped files?

A memory-mapped file is a file with data that the operating system places in memory by way of the `mmap()` system call. `mmap()` thus *maps* the file to a region of virtual memory. Memory-mapped files are the critical piece of the storage engine in MongoDB. By using memory mapped files MongoDB can treat the contents of its data files as if they were in memory. This provides MongoDB with an extremely fast and simple method for accessing and manipulating data.

11.7.2 How do memory mapped files work?

Memory mapping assigns files to a block of virtual memory with a direct byte-for-byte correlation. Once mapped, the relationship between file and memory allows MongoDB to interact with the data in the file as if it were memory.

11.7.3 How does MongoDB work with memory mapped files?

MongoDB uses memory mapped files for managing and interacting with all data. MongoDB memory maps data files to memory as it accesses documents. Data that isn't accessed is *not* mapped to memory.

11.7.4 What are page faults?

Page faults can occur as MongoDB reads from or writes data to parts of its data files that are not currently located in physical memory. In contrast, operating system page faults happen when physical memory is exhausted and pages of physical memory are swapped to disk.

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.

[27] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

This process, particularly on an active system can take a long time, particularly in comparison to reading a page that is already in memory.

See *Page Faults* (page 229) for more information.

11.7.5 What is the difference between soft and hard page faults?

Page faults occur when MongoDB needs access to data that isn't currently in active memory. A "hard" page fault refers to situations when MongoDB must access a disk to access the data. A "soft" page fault, by contrast, merely moves memory pages from one list to another, such as from an operating system file cache. In production, MongoDB will rarely encounter soft page faults.

See *Page Faults* (page 229) for more information.

11.7.6 What tools can I use to investigate storage use in MongoDB?

The `db.stats()` method in the `mongo` shell, returns the current state of the "active" database. The `dbStats command` document describes the fields in the `db.stats()` output.

11.7.7 What is the working set?

Working set represents the total body of data that the application uses in the course of normal operation. Often this is a subset of the total data size, but the specific size of the working set depends on actual moment-to-moment use of the database.

If you run a query that requires MongoDB to scan every document in a collection, the working set will expand to include every document. Depending on physical memory size, this may cause documents in the working set to "page out," or to be removed from physical memory by the operating system. The next time MongoDB needs to access these documents, MongoDB may incur a hard page fault.

If you run a query that requires MongoDB to scan every *document* in a collection, the working set includes every active document in memory.

For best performance, the majority of your *active* set should fit in RAM.

11.7.8 Why are the files in my data directory larger than the data in my database?

The data files in your data directory, which is the `/data/db` directory in default configurations, might be larger than the data set inserted into the database. Consider the following possible causes:

Preallocated data files

In the data directory, MongoDB preallocates data files to a particular size, in part to prevent file system fragmentation. MongoDB names the first data file `<databasename>.0`, the next `<databasename>.1`, etc. The first file `mongod` allocates is 64 megabytes, the next 128 megabytes, and so on, up to 2 gigabytes, at which point all subsequent files are 2 gigabytes. The data files include files with allocated space but that hold no data. `mongod` may allocate a 1 gigabyte data file that may be 90% empty. For most larger databases, unused allocated space is small compared to the database.

The `oplog`

If this `mongod` is a member of a replica set, the data directory includes the *oplog.rs* file, which is a preallocated *capped collection* in the `local` database.

The default allocation is approximately 5% of disk space on 64-bit installations. In most cases, you should not need to resize the oplog. See *Oplog Sizing* (page 597) for more information

The `journal`

The data directory contains the journal files, which store write operations on disk before MongoDB applies them to databases. See *Journaling Mechanics* (page 309).

Empty records

MongoDB maintains lists of empty records in data files as it deletes documents and collections. MongoDB can reuse this space, but will not, by default, return this space to the operating system.

To allow MongoDB to more effectively reuse the space, you can de-fragment your data. To de-fragment, use the `compact` command. The `compact` requires up to 2 gigabytes of extra disk space to run. Do not use `compact` if you are critically low on disk space. For more information on its behavior and other considerations, see `compact`.

`compact` only removes fragmentation from MongoDB data files within a collection and does not return any disk space to the operating system. To return disk space to the operating system, see *How do I reclaim disk space?* (page 795).

11.7.9 How do I reclaim disk space?

The following provides some options to consider when reclaiming disk space.

Note: You do not need to reclaim disk space for MongoDB to reuse freed space. See *Empty records* (page 795) for information on reuse of freed space.

`repairDatabase`

You can use `repairDatabase` on a database to rebuilds the database, de-fragmenting the associated storage in the process.

`repairDatabase` requires free disk space equal to the size of your current data set plus 2 gigabytes. If the volume that holds dbpath lacks sufficient space, you can mount a separate volume and use that for the repair. For additional information and considerations, see `repairDatabase`.

> **Warning:** Do not use `repairDatabase` if you are critically low on disk space.
> `repairDatabase` will block all other operations and may take a long time to complete.

You can only run `repairDatabase` on a standalone `mongod` instance.

You can also run the `repairDatabase` operation for all databases on the server by restarting your `mongod` standalone instance with the `--repair` and `--repairpath` options. All databases on the server will be unavailable during this operation.

Resync the Member of the Replica Set

For a secondary member of a replica set, you can perform a *resync of the member* (page 640) by: stopping the secondary member to resync, deleting all data and subdirectories from the member's data directory, and restarting.

For details, see *Resync a Member of a Replica Set* (page 640).

11.7.10 How can I check the size of a collection?

To view the size of a collection and other information, use the `db.collection.stats()` method from the `mongo` shell. The following example issues `db.collection.stats()` for the `orders` collection:

```
db.orders.stats();
```

To view specific measures of size, use these methods:

- `db.collection.dataSize()`: data size in bytes for the collection.
- `db.collection.storageSize()`: allocation size in bytes, including unused space.
- `db.collection.totalSize()`: the data size plus the index size in bytes.
- `db.collection.totalIndexSize()`: the index size in bytes.

Also, the following scripts print the statistics for each database and collection:

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); prin
```

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); mdb
```

11.7.11 How can I check the size of indexes?

To view the size of the data allocated for an index, use one of the following procedures in the `mongo` shell:

- Use the `db.collection.stats()` method using the index namespace. To retrieve a list of namespaces, issue the following command:

  ```
  db.system.namespaces.find()
  ```

- Check the value of `indexSizes` in the output of the `db.collection.stats()` command.

Example

Issue the following command to retrieve index namespaces:

```
db.system.namespaces.find()
```

The command returns a list similar to the following:

```
{"name" : "test.orders"}
{"name" : "test.system.indexes"}
{"name" : "test.orders.$_id_"}
```

View the size of the data allocated for the `orders.$_id_` index with the following sequence of operations:

```
use test
db.orders.$_id_.stats().indexSizes
```

11.7.12 How do I know when the server runs out of disk space?

If your server runs out of disk space for data files, you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:09 [FileAllocator]     will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:19 [FileAllocator]     will try again in 10 seconds
```

The server remains in this state forever, blocking all writes including deletes. However, reads still work. To delete some data and compact, using the `compact` command, you must restart the server first.

If your server runs out of disk space for journal files, the server process will exit. By default, `mongod` creates journal files in a sub-directory of `dbPath` named `journal`. You may elect to put the journal files on another storage device using a filesystem mount or a symlink.

Note: If you place the journal files on a separate storage device you will not be able to use a file system snapshot tool to capture a valid snapshot of your data files and journal files.

11.8 FAQ: Indexes

On this page

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 761) or post your question to the MongoDB User Mailing List[28]. See also *Indexing Tutorials* (page 519).

11.8.1 Should you run `ensureIndex()` after every insert?

No. You only need to create an index once for a single collection. After initial creation, MongoDB automatically updates the index as data changes.

While running `ensureIndex()` is usually ok, if an index doesn't exist because of ongoing administrative work, a call to `ensureIndex()` may disrupt database availability. Running `ensureIndex()` can render a replica set inaccessible as the index creation is happening. See *Build Indexes on Replica Sets* (page 524).

[28] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

11.8.2 How do you know what indexes exist in a collection?

To list a collection's indexes, use the `db.collection.getIndexes()` method or a similar method for your driver[29].

11.8.3 How do you determine the size of an index?

To check the sizes of the indexes on a collection, use `db.collection.stats()`.

11.8.4 What happens if an index does not fit into RAM?

When an index is too large to fit into RAM, MongoDB must read the index from disk, which is a much slower operation than reading from RAM. Keep in mind an index fits into RAM when your server has RAM available for the index combined with the rest of the *working set*.

In certain cases, an index does not need to fit *entirely* into RAM. For details, see *Indexes that Hold Only Recent Values in RAM* (page 555).

11.8.5 How do you know what index a query used?

To inspect how MongoDB processes a query, use the `explain()` method in the `mongo` shell, or in your application driver.

11.8.6 How do you determine what fields to index?

A number of factors determine what fields to index, including *selectivity* (page 555), fitting indexes into RAM, reusing indexes in multiple queries when possible, and creating indexes that can support all the fields in a given query. For detailed documentation on choosing which fields to index, see *Indexing Tutorials* (page 519).

11.8.7 How do write operations affect indexes?

Any write operation that alters an indexed field requires an update to the index in addition to the document itself. If you update a document that causes the document to grow beyond the allotted record size, then MongoDB must update all indexes that include this document as part of the update operation.

Therefore, if your application is write-heavy, creating too many indexes might affect performance.

11.8.8 Will building a large index affect database performance?

Building an index can be an IO-intensive operation, especially if you have a large collection. This is true on any database system that supports secondary indexes, including MySQL. If you need to build an index on a large collection, consider building the index in the background. See *Index Creation* (page 509).

If you build a large index without the background option, and if doing so causes the database to stop responding, do one of the following:

- Wait for the index to finish building.
- Kill the current operation (see `db.killOp()`). The partially built index will be deleted.

[29] https://api.mongodb.org/

11.8.9 Can I use index keys to constrain query matches?

You can use the `min()` and `max()` methods to constrain the results of the cursor returned from `find()` by using index keys.

11.8.10 Using `$ne` and `$nin` in a query is slow. Why?

The `$ne` and `$nin` operators are not selective. See *Create Queries that Ensure Selectivity* (page 555). If you need to use these, it is often best to make sure that an additional, more selective criterion is part of the query.

11.8.11 Can I use a multi-key index to support a query for a whole array?

Not entirely. The index can partially support these queries because it can speed the selection of the first element of the array; however, comparing all subsequent items in the array cannot use the index and must scan the documents individually.

11.8.12 How can I effectively use indexes strategy for attribute lookups?

For simple attribute lookups that don't require sorted result sets or range queries, consider creating a field that contains an array of documents where each document has a field (e.g. `attrib`) that holds a specific type of attribute. You can index this `attrib` field.

For example, the `attrib` field in the following document allows you to add an unlimited number of attributes types:

```
{ _id : ObjectId(...),
  attrib : [
             { k: "color", v: "red" },
             { k: "shape": v: "rectangle" },
             { k: "color": v: "blue" },
             { k: "avail": v: true }
           ]
}
```

Both of the following queries could use the same `{ "attrib.k": 1, "attrib.v": 1 }` index:

```
db.mycollection.find( { attrib: { $elemMatch : { k: "color", v: "blue" } } } )
db.mycollection.find( { attrib: { $elemMatch : { k: "avail", v: true } } } )
```

11.9 FAQ: MongoDB Diagnostics

On this page

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 761) or post your question to the MongoDB User Mailing List[30].

11.9.1 Where can I find information about a `mongod` process that stopped running unexpectedly?

If `mongod` shuts down unexpectedly on a UNIX or UNIX-based platform, and if `mongod` fails to log a shutdown or error message, then check your system logs for messages pertaining to MongoDB. For example, for logs located in `/var/log/messages`, use the following commands:

```
sudo grep mongod /var/log/messages
sudo grep score /var/log/messages
```

11.9.2 Does TCP `keepalive` time affect sharded clusters and replica sets?

If you experience socket errors between members of a sharded cluster or replica set, that do not have other reasonable causes, check the TCP keep alive value, which Linux systems store as the `tcp_keepalive_time` value. A common keep alive period is `7200` seconds (2 hours); however, different distributions and OS X may have different settings. For MongoDB, you will have better experiences with shorter keepalive periods, on the order of `300` seconds (five minutes).

On Linux systems you can use the following operation to check the value of `tcp_keepalive_time`:

```
cat /proc/sys/net/ipv4/tcp_keepalive_time
```

You can change the `tcp_keepalive_time` value with the following operation:

```
echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time
```

The new `tcp_keepalive_time` value takes effect without requiring you to restart the `mongod` or `mongos` servers. When you reboot or restart your system you will need to set the new `tcp_keepalive_time` value, or see your operating system's documentation for setting the TCP keepalive value persistently.

For OS X systems, issue the following command to view the keep alive setting:

```
sysctl net.inet.tcp.keepinit
```

To set a shorter keep alive period use the following invocation:

```
sysctl -w net.inet.tcp.keepinit=300
```

If your replica set or sharded cluster experiences keepalive-related issues, you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This includes all machines hosting `mongos` or `mongod` servers.

Windows users should consider the Windows Server Technet Article on KeepAliveTime configuration[31] for more information on setting keep alive for MongoDB deployments on Windows systems.

11.9.3 What tools are available for monitoring MongoDB?

The MongoDB Cloud Manager[32] and Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[33] include monitoring functionality, which collects data from running MongoDB deployments and provides

[30] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
[31] http://technet.microsoft.com/en-us/library/dd349797.aspx#BKMK_2
[32] https://cloud.mongodb.com/?jmp=docs
[33] https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

visualization and alerts based on that data.

For more information, see also the MongoDB Cloud Manager documentation[34] and Ops Manager documentation[35].

A full list of third-party tools is available as part of the *Monitoring for MongoDB* (page 195) documentation.

11.9.4 Memory Diagnostics

Do I need to configure swap space?

Always configure systems to have swap space. Without swap, your system may not be reliant in some situations with extreme memory constraints, memory leaks, or multiple programs using the same memory. Think of the swap space 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 793) and should constitute most of your MongoDB memory use. Therefore, it is unlikely that `mongod` will ever use any swap space in normal operation. The operating system will release memory from the memory mapped files without needing swap and MongoDB can write data to the data files without needing the swap system.

What is "working set" and how can I estimate its size?

The *working set* for a MongoDB database is the portion of your data that clients access most often. You can estimate size of the working set, using the `workingSet` document in the output of `serverStatus`. To return `serverStatus` with the `workingSet` document, issue a command in the following form:

```
db.runCommand( { serverStatus: 1, workingSet: 1 } )
```

Must my working set size fit RAM?

Your working set should stay in memory to achieve good performance. Otherwise many random disk IO's will occur, and unless you are using SSD, this can be quite slow.

One area to watch specifically in managing the size of your working set is index access patterns. If you are inserting into indexes at random locations (as would happen with id's that are randomly generated by hashes), you will continually be updating the whole index. If instead you are able to create your id's in approximately ascending order (for example, day concatenated with a random id), all the updates will occur at the right side of the b-tree and the working set size for index pages will be much smaller.

It is fine if databases and thus virtual size are much larger than RAM.

How do I calculate how much RAM I need for my application?

The amount of RAM you need depends on several factors, including but not limited to:

- The relationship between *database storage* (page 792) and working set.
- The operating system's cache strategy for LRU (Least Recently Used)
- The impact of *journaling* (page 309)
- The number or rate of page faults and other MongoDB Cloud Manager gauges to detect when you need more RAM

[34] https://docs.cloud.mongodb.com/
[35] https://docs.opsmanager.mongodb.com/current/application

- Each database connection thread will need up to 1 MB of RAM.

MongoDB defers to the operating system when loading data into memory from disk. It simply *memory maps* (page 793) 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. Because MongoDB uses a thread per connection model, each database connection also will need up to 1MB of RAM, whether active or idle.

If page faults are infrequent, your working set fits in RAM. If fault rates rise higher than that, you risk performance degradation. This is less critical with SSD drives than with spinning disks.

How do I read memory statistics in the UNIX `top` command

Because `mongod` uses *memory-mapped files* (page 793), the memory statistics in `top` require interpretation in a special way. On a large database, `VSIZE` (virtual bytes) tends to be the size of the entire database. If the `mongod` doesn't have other processes running, `RSIZE` (resident bytes) is the total memory of the machine, as this counts file system cache contents.

For Linux systems, use the `vmstat` command to help determine how the system uses memory. On OS X systems use `vm_stat`.

11.9.5 Sharded Cluster Diagnostics

The two most important factors in maintaining a successful sharded cluster are:

- *choosing an appropriate shard key* (page 687) and
- *sufficient capacity to support current and future operations* (page 685).

You can prevent most issues encountered with sharding by ensuring that you choose the best possible *shard key* for your deployment and ensure that you are always adding additional capacity to your cluster well before the current resources become saturated. Continue reading for specific issues you may encounter in a production environment.

In a new sharded cluster, why does all data remains on one shard?

Your cluster must have sufficient data for sharding to make sense. Sharding works by migrating chunks between the shards until each shard has roughly the same number of chunks.

The default chunk size is 64 megabytes. MongoDB will not begin migrations until the imbalance of chunks in the cluster exceeds the *migration threshold* (page 699). While the default chunk size is configurable with the `chunkSize` setting, these behaviors help prevent unnecessary chunk migrations, which can degrade the performance of your cluster as a whole.

If you have just deployed a sharded cluster, make sure that you have enough data to make sharding effective. If you do not have sufficient data to create more than eight 64 megabyte chunks, then all data will remain on one shard. Either lower the *chunk size* (page 702) 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 738).

Finally, if your shard key has a low *cardinality* (page 710), MongoDB may not be able to create sufficient splits among the data.

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 689).

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 709) to correct this pattern.

What can prevent a sharded cluster from balancing?

If you have just deployed your sharded cluster, you may want to consider the *troubleshooting suggestions for a new cluster where data remains on a single shard* (page 802).

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 710) 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 731) that is too short, given the rate of data growth.
  - an uneven distribution of *write operations* (page 689) that requires more data migration. You may have to choose a different shard key to resolve this issue.
  - poor network connectivity between shards, which may lead to chunk migrations that take too long to complete. Investigate your network configuration and interconnections between shards.

Why do chunk migrations affect sharded cluster performance?

If migrations impact your cluster or application's performance, consider the following options, depending on the nature of the impact:

1. If migrations only interrupt your clusters sporadically, you can limit the *balancing window* (page 731) 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 743) 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 712) 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 689).

CHAPTER 12

Release Notes

Always install the latest, stable version of MongoDB. See *MongoDB Version Numbers* (page 908) for more information.

See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

12.1 Current Stable Release

(*2.6-series*)

12.1.1 Release Notes for MongoDB 2.6

On this page

April 8, 2014

MongoDB 2.6 is now available. Key features include aggregation enhancements, text-search integration, query-engine improvements, a new write-operation protocol, and security enhancements.

Minor Releases

2.6 Changelog

On this page

2.6.11 – Changes

Querying

- SERVER-19553[1] `mongod` shouldn't use `sayPiggyBack` to send `killCursor` messages
- SERVER-18620[2] Reduce frequency of "`staticYield can't unlock`" log message
- SERVER-18461[3] Range predicates comparing against a BinData value should be covered, but are not in 2.6
- SERVER-17815[4] Plan ranking tie breaker is computed incorrectly
- SERVER-16265[5] Add query details to getmore entry in profiler and `db.currentOp()`
- SERVER-15217[6] v2.6 query plan ranking test "`NonCoveredIxisectFetchesLess`" relies on order of deleted record list
- SERVER-14070[7] Compound index not providing sort if equality predicate given on sort field

Replication

- SERVER-18280[8] `ReplicaSetMonitor` should use `electionId` to avoid talking to old primaries
- SERVER-18795[9] `db.printSlaveReplicationInfo()`/`rs.printSlaveReplicationInfo()` can not work with `ARBITER` role

Sharding

- SERVER-19464[10] `$sort` stage in aggregation doesn't call scoped connections done ()
- SERVER-18955[11] `mongos` doesn't set batch size (and keeps the old one, 0) on getMore if performed on first `_cursor->more()`

[1] https://jira.mongodb.org/browse/SERVER-19553
[2] https://jira.mongodb.org/browse/SERVER-18620
[3] https://jira.mongodb.org/browse/SERVER-18461
[4] https://jira.mongodb.org/browse/SERVER-17815
[5] https://jira.mongodb.org/browse/SERVER-16265
[6] https://jira.mongodb.org/browse/SERVER-15217
[7] https://jira.mongodb.org/browse/SERVER-14070
[8] https://jira.mongodb.org/browse/SERVER-18280
[9] https://jira.mongodb.org/browse/SERVER-18795
[10] https://jira.mongodb.org/browse/SERVER-19464
[11] https://jira.mongodb.org/browse/SERVER-18955

Indexing

- SERVER-19559[12] Document growth of "key too large" document makes it disappear from the index
- SERVER-16348[13] `Assertion failure n >= 0 && n < static_cast<int>(_files.size()) src/mongo/db/storage/extent_manager.cpp 109`
- SERVER-13875[14] `ensureIndex()` of `2dsphere` index breaks after upgrading to 2.6 (with the new `createIndex` command)

Networking SERVER-19389[15] Remove wire level endianness check

Build and Testing

- SERVER-18097[16] Remove `mongosTest_auth` and `mongosTest_WT` tasks from `evergreen.yml`
- SERVER-18068[17] Coverity analysis defect 72413: Resource leak
- SERVER-18371[18] Add SSL library config detection

2.6.10 – Changes

Security

- SERVER-18312[19] Upgrade PCRE to latest
- SERVER-17812[20] LockPinger has audit-related GLE failure
- SERVER-17647[21] Compute BinData length in v8
- SERVER-17591[22] Add SSL flag to select supported protocols
- SERVER-16849[23] On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-11980[24] Improve user cache invalidation enforcement on mongos

Querying

- SERVER-18364[25] Ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction
- SERVER-17815[26] Plan ranking tie breaker is computed incorrectly
- SERVER-16256[27] $all clause with elemMatch uses wider bounds than needed

[12] https://jira.mongodb.org/browse/SERVER-19559
[13] https://jira.mongodb.org/browse/SERVER-16348
[14] https://jira.mongodb.org/browse/SERVER-13875
[15] https://jira.mongodb.org/browse/SERVER-19389
[16] https://jira.mongodb.org/browse/SERVER-18097
[17] https://jira.mongodb.org/browse/SERVER-18068
[18] https://jira.mongodb.org/browse/SERVER-18371
[19] https://jira.mongodb.org/browse/SERVER-18312
[20] https://jira.mongodb.org/browse/SERVER-17812
[21] https://jira.mongodb.org/browse/SERVER-17647
[22] https://jira.mongodb.org/browse/SERVER-17591
[23] https://jira.mongodb.org/browse/SERVER-16849
[24] https://jira.mongodb.org/browse/SERVER-11980
[25] https://jira.mongodb.org/browse/SERVER-18364
[26] https://jira.mongodb.org/browse/SERVER-17815
[27] https://jira.mongodb.org/browse/SERVER-16256

Replication

- SERVER-18211[28] MongoDB fails to correctly roll back collection creation
- SERVER-17771[29] Reconfiguring a replica set to remove a node causes a segmentation fault on 2.6.8
- SERVER-13542[30] Expose electionId on primary in isMaster

Sharding

- SERVER-17812[31] LockPinger has audit-related GLE failure
- SERVER-17805[32] logOp / OperationObserver should always check shardversion
- SERVER-17749[33] `collMod usePowerOf2Sizes` fails on `mongos`
- SERVER-11980[34] Improve user cache invalidation enforcement on `mongos`

Storage

- SERVER-18211[35] MongoDB fails to correctly roll back collection creation
- SERVER-17653[36] ERROR: socket XXX is higher than 1023; not supported on 2.6.*

Indexing SERVER-17018[37] Assertion failure false `src/mongo/db/structure/btree/key.cpp` Line 433 on remove operation

Write Ops

- SERVER-18111[38] `mongod` allows user inserts into `system.profile` collection
- SERVER-13542[39] Expose `electionId` on primary in `isMaster`

Networking

- SERVER-18096[40] Shard primary incorrectly reuses closed sockets after relinquish and re-election
- SERVER-17591[41] Add SSL flag to select supported protocols

Build and Packaging

- SERVER-18344[42] logs should be sent to updated logkeeper server
- SERVER-18082[43] Change smoke.py buildlogger command line options to environment variables

[28] https://jira.mongodb.org/browse/SERVER-18211
[29] https://jira.mongodb.org/browse/SERVER-17771
[30] https://jira.mongodb.org/browse/SERVER-13542
[31] https://jira.mongodb.org/browse/SERVER-17812
[32] https://jira.mongodb.org/browse/SERVER-17805
[33] https://jira.mongodb.org/browse/SERVER-17749
[34] https://jira.mongodb.org/browse/SERVER-11980
[35] https://jira.mongodb.org/browse/SERVER-18211
[36] https://jira.mongodb.org/browse/SERVER-17653
[37] https://jira.mongodb.org/browse/SERVER-17018
[38] https://jira.mongodb.org/browse/SERVER-18111
[39] https://jira.mongodb.org/browse/SERVER-13542
[40] https://jira.mongodb.org/browse/SERVER-18096
[41] https://jira.mongodb.org/browse/SERVER-17591
[42] https://jira.mongodb.org/browse/SERVER-18344
[43] https://jira.mongodb.org/browse/SERVER-18082

- SERVER-18312[44] Upgrade PCRE to latest
- SERVER-17780[45] Init script sets process ulimit to different value compared to documentation
- SERVER-16563[46] Debian repo component mismatch - mongodb/10gen

Shell SERVER-17951[47] db.currentOp() fails with read preference set

Testing

- SERVER-18262[48] setup_multiversion_mongodb should retry links download on timeouts
- SERVER-18229[49] `smoke.py` with PyMongo 3.0.1 fails to run certain tests
- SERVER-18073[50] Fix `smoke.py` to work with PyMongo 3.0

2.6.9 – Changes

Security SERVER-16073[51] Create hidden `net.ssl.sslCipherConfig` flag

Querying

- SERVER-14723[52] Crash during query planning for `geoNear` with multiple `2dsphere` indexes
- SERVER-14071[53] For queries with `sort()`, bad non-blocking plan can be cached if there are zero results
- SERVER-8188[54] Configurable idle cursor timeout

Replication and Sharding

- SERVER-17429[55] the message logged when changing sync target due to stale data should format OpTimes in a consistent way
- SERVER-17441[56] `mongos` crash right after "not master" error

Storage SERVER-15907[57] Use `ftruncate` rather than `fallocate` when running on `tmpfs`

[44] https://jira.mongodb.org/browse/SERVER-18312
[45] https://jira.mongodb.org/browse/SERVER-17780
[46] https://jira.mongodb.org/browse/SERVER-16563
[47] https://jira.mongodb.org/browse/SERVER-17951
[48] https://jira.mongodb.org/browse/SERVER-18262
[49] https://jira.mongodb.org/browse/SERVER-18229
[50] https://jira.mongodb.org/browse/SERVER-18073
[51] https://jira.mongodb.org/browse/SERVER-16073
[52] https://jira.mongodb.org/browse/SERVER-14723
[53] https://jira.mongodb.org/browse/SERVER-14071
[54] https://jira.mongodb.org/browse/SERVER-8188
[55] https://jira.mongodb.org/browse/SERVER-17429
[56] https://jira.mongodb.org/browse/SERVER-17441
[57] https://jira.mongodb.org/browse/SERVER-15907

Aggregation Framework

- SERVER-17426[58] Aggregation framework query by `_id` returns duplicates in sharded cluster (orphan documents)
- SERVER-17224[59] Aggregation pipeline with 64MB document can terminate server

Build and Platform

- SERVER-17484[60] Migrate server MCI config into server repo
- SERVER-17252[61] Upgrade PCRE Version from 8.30 to Latest

Diagnostics and Internal Code

- SERVER-17226[62] `top` command with 64MB result document can terminate server
- SERVER-17338[63] NULL pointer crash when running `copydb` against stepped-down 2.6 primary
- SERVER-14992[64] Query for Windows 7 File Allocation Fix, and other hotfixes

2.6.8 – Changes

Security and Networking

- SERVER-17278[65] BSON BinData validation enforcement
- SERVER-17022[66] No SSL Session Caching may not be respected
- SERVER-17264[67] improve bson validation

Query and Aggregation

- SERVER-16655[68] Geo predicate is unable to use compound 2dsphere index if it is root of `$or` clause
- SERVER-16527[69] `2dsphere` explain reports "works" for `nscanned` & `nscannedObjects`
- SERVER-15802[70] Query optimizer should always use equality predicate over unique index when possible
- SERVER-14044[71] Incorrect `{$meta: 'text'}` reference in aggregation `$sort` error message

[58] https://jira.mongodb.org/browse/SERVER-17426
[59] https://jira.mongodb.org/browse/SERVER-17224
[60] https://jira.mongodb.org/browse/SERVER-17484
[61] https://jira.mongodb.org/browse/SERVER-17252
[62] https://jira.mongodb.org/browse/SERVER-17226
[63] https://jira.mongodb.org/browse/SERVER-17338
[64] https://jira.mongodb.org/browse/SERVER-14992
[65] https://jira.mongodb.org/browse/SERVER-17278
[66] https://jira.mongodb.org/browse/SERVER-17022
[67] https://jira.mongodb.org/browse/SERVER-17264
[68] https://jira.mongodb.org/browse/SERVER-16655
[69] https://jira.mongodb.org/browse/SERVER-16527
[70] https://jira.mongodb.org/browse/SERVER-15802
[71] https://jira.mongodb.org/browse/SERVER-14044

Replication

- SERVER-16599[72] `copydb` and `clone` commands can crash the server if a primary steps down
- SERVER-16315[73] Replica set nodes should not threaten to veto nodes whose config version is higher than their own
- SERVER-16274[74] secondary `fasserts` trying to replicate an index
- SERVER-15471[75] Better error message when replica is not found in `GhostSync::associateSlave`

Sharding

- SERVER-17191[76] Spurious warning during upgrade of sharded cluster
- SERVER-17163[77] Fatal error "logOp but not primary" in `MigrateStatus::go`
- SERVER-16984[78] `UpdateLifecycleImpl` can return empty `collectionMetadata` even if `ns` is sharded
- SERVER-10904[79] Possible for `_master` and `_slaveConn` to be pointing to different connections even with primary read pref

Storage

- SERVER-17087[80] Add listCollections command functionality to 2.6 shell & client
- SERVER-14572[81] Increase C runtime stdio file limit

Tools

- SERVER-17216[82] 2.6 `mongostat` cannot be used with 3.0 `mongod`
- SERVER-14190[83] `mongorestore parseMetadataFile` passes non-null terminated string to ‘`fromjson`‘

Build and Packaging

- SERVER-14803[84] Support static libstdc++ builds for non-Linux builds
- SERVER-15400[85] Create Windows Enterprise Zip File with vcredist and dependent dlls

Usability SERVER-14756[86] The YAML `storage.quota.enforced` option is not found

[72] https://jira.mongodb.org/browse/SERVER-16599
[73] https://jira.mongodb.org/browse/SERVER-16315
[74] https://jira.mongodb.org/browse/SERVER-16274
[75] https://jira.mongodb.org/browse/SERVER-15471
[76] https://jira.mongodb.org/browse/SERVER-17191
[77] https://jira.mongodb.org/browse/SERVER-17163
[78] https://jira.mongodb.org/browse/SERVER-16984
[79] https://jira.mongodb.org/browse/SERVER-10904
[80] https://jira.mongodb.org/browse/SERVER-17087
[81] https://jira.mongodb.org/browse/SERVER-14572
[82] https://jira.mongodb.org/browse/SERVER-17216
[83] https://jira.mongodb.org/browse/SERVER-14190
[84] https://jira.mongodb.org/browse/SERVER-14803
[85] https://jira.mongodb.org/browse/SERVER-15400
[86] https://jira.mongodb.org/browse/SERVER-14756

Testing SERVER-16421[87] `sharding_rs2.js` should clean up data on all replicas

2.6.7 – Changes

Stability

- SERVER-16237[88] Don't check the shard version if the primary server is down

Querying

- SERVER-16408[89] `max_time_ms.js` should not run in parallel suite.

Replication

- SERVER-16732[90] `SyncSourceFeedback::replHandshake()` may perform an illegal erase from a `std::map` in some circumstances

Sharding

- SERVER-16683[91] Decrease mongos memory footprint when shards have several tags
- SERVER-15766[92] prefix_shard_key.js depends on primary allocation to particular shards
- SERVER-14306[93] `mongos` can cause shards to hit the in-memory sort limit by requesting more results than needed.

Packaging

- SERVER-16081[94] `/etc/init.d/mongod` startup script fails, with dirname message

2.6.6 – Changes

Security

- SERVER-15673[95] Disable SSLv3 ciphers
- SERVER-15515[96] New test for mixed version replSet, 2.4 primary, user updates
- SERVER-15500[97] New test for system.user operations

[87] https://jira.mongodb.org/browse/SERVER-16421
[88] https://jira.mongodb.org/browse/SERVER-16237
[89] https://jira.mongodb.org/browse/SERVER-16408
[90] https://jira.mongodb.org/browse/SERVER-16732
[91] https://jira.mongodb.org/browse/SERVER-16683
[92] https://jira.mongodb.org/browse/SERVER-15766
[93] https://jira.mongodb.org/browse/SERVER-14306
[94] https://jira.mongodb.org/browse/SERVER-16081
[95] https://jira.mongodb.org/browse/SERVER-15673
[96] https://jira.mongodb.org/browse/SERVER-15515
[97] https://jira.mongodb.org/browse/SERVER-15500

Stability

- SERVER-12061[98] Do not silently ignore read errors when syncing a replica set node
- SERVER-12058[99] Primary should abort if encountered problems writing to the oplog

Querying

- SERVER-16291[100] Cannot set/list/clear index filters on the secondary
- SERVER-15958[101] The "isMultiKey" value is not correct in the output of aggregation explain plan
- SERVER-15899[102] Querying against path in document containing long array of subdocuments with nested arrays causes stack overflow
- SERVER-15696[103] `$regex`, `$in` and `$sort` with index returns too many results
- SERVER-15639[104] Text queries can return incorrect results and leak memory when multiple predicates given on same text index prefix field
- SERVER-15580[105] Evaluating candidate query plans with concurrent writes on same collection may crash `mongod`
- SERVER-15528[106] Distinct queries can scan many index keys without yielding read lock
- SERVER-15485[107] CanonicalQuery::canonicalize can leak a LiteParsedQuery
- SERVER-15403[108] `$min` and `$max` equal errors in 2.6 but not in 2.4
- SERVER-15233[109] Cannot run `planCacheListQueryShapes` on a Secondary
- SERVER-14799[110] `count` with hint doesn't work when hint is a document

Replication

- SERVER-16107[111] 2.6 `mongod` crashes with segfault when added to a 2.8 replica set with >= 12 nodes.
- SERVER-15994[112] `listIndexes` and `listCollections` can be run on secondaries without slaveOk bit
- SERVER-15849[113] do not forward replication progress for nodes that are no longer part of a replica set
- SERVER-15491[114] `SyncSourceFeedback` can crash due to a `SocketException` in `authenticateInternalUser`

[98] https://jira.mongodb.org/browse/SERVER-12061
[99] https://jira.mongodb.org/browse/SERVER-12058
[100] https://jira.mongodb.org/browse/SERVER-16291
[101] https://jira.mongodb.org/browse/SERVER-15958
[102] https://jira.mongodb.org/browse/SERVER-15899
[103] https://jira.mongodb.org/browse/SERVER-15696
[104] https://jira.mongodb.org/browse/SERVER-15639
[105] https://jira.mongodb.org/browse/SERVER-15580
[106] https://jira.mongodb.org/browse/SERVER-15528
[107] https://jira.mongodb.org/browse/SERVER-15485
[108] https://jira.mongodb.org/browse/SERVER-15403
[109] https://jira.mongodb.org/browse/SERVER-15233
[110] https://jira.mongodb.org/browse/SERVER-14799
[111] https://jira.mongodb.org/browse/SERVER-16107
[112] https://jira.mongodb.org/browse/SERVER-15994
[113] https://jira.mongodb.org/browse/SERVER-15849
[114] https://jira.mongodb.org/browse/SERVER-15491

Sharding

- SERVER-15318[115] `copydb` should not use exhaust flag when used against `mongos`
- SERVER-14728[116] Shard depends on string comparison of replica set connection string
- SERVER-14506[117] special top chunk logic can move max chunk to a shard with incompatible tag
- SERVER-14299[118] For sharded limit=N queries with sort, mongos can request >N results from shard
- SERVER-14080[119] Have migration result reported in the changelog correctly
- SERVER-12472[120] Fail MoveChunk if an index is needed on TO shard and data exists

Storage

- SERVER-16283[121] Can't start new wiredtiger node with log file or config file in data directory - false detection of old `mmapv1` files
- SERVER-15986[122] Starting with different storage engines in the same dbpath should error/warn
- SERVER-14057[123] Changing TTL expiration time with collMod does not correctly update index definition

Indexing and write Operations

- SERVER-14287[124] ensureIndex can abort reIndex and lose indexes
- SERVER-14886[125] Updates against paths composed with array index notation and positional operator fail with error

Data Aggregation SERVER-15552[126] Errors writing to temporary collections during `mapReduce` command execution should be operation-fatal

Build and Packaging

- SERVER-14184[127] Unused preprocessor macros from s2 conflict on OS X Yosemite
- SERVER-14015[128] S2 Compilation on GCC 4.9/Solaris fails
- SERVER-16017[129] Suse11 enterprise packages fail due to unmet dependencies
- SERVER-15598[130] Ubuntu 14.04 Enterprise packages depend on unavailable libsnmp15 package
- SERVER-13595[131] Red Hat init.d script error: YAML config file parsing

[115] https://jira.mongodb.org/browse/SERVER-15318
[116] https://jira.mongodb.org/browse/SERVER-14728
[117] https://jira.mongodb.org/browse/SERVER-14506
[118] https://jira.mongodb.org/browse/SERVER-14299
[119] https://jira.mongodb.org/browse/SERVER-14080
[120] https://jira.mongodb.org/browse/SERVER-12472
[121] https://jira.mongodb.org/browse/SERVER-16283
[122] https://jira.mongodb.org/browse/SERVER-15986
[123] https://jira.mongodb.org/browse/SERVER-14057
[124] https://jira.mongodb.org/browse/SERVER-14287
[125] https://jira.mongodb.org/browse/SERVER-14886
[126] https://jira.mongodb.org/browse/SERVER-15552
[127] https://jira.mongodb.org/browse/SERVER-14184
[128] https://jira.mongodb.org/browse/SERVER-14015
[129] https://jira.mongodb.org/browse/SERVER-16017
[130] https://jira.mongodb.org/browse/SERVER-15598
[131] https://jira.mongodb.org/browse/SERVER-13595

Logging and Diagnostics

- SERVER-13471[132] Increase log level of "did reduceInMemory" message in map/reduce
- SERVER-16324[133] Command execution log line displays "`query not recording (too large)`" instead of abbreviated command object
- SERVER-10069[134] Improve errorcodes.py so it captures multiline messages

Testing and Internals

- SERVER-15632[135] `MultiHostQueryOp::PendingQueryContext::doBlockingQuery` can leak a cursor object
- SERVER-15629[136] `GeoParser::parseMulti{Line|Polygon}` does not clear objects owned by out parameter
- SERVER-16316[137] Remove unsupported behavior in shard3.js
- SERVER-14763[138] Update jstests/sharding/split_large_key.js
- SERVER-14249[139] Add tests for querying oplog via mongodump using –dbpath
- SERVER-13726[140] indexbg_drop.js

2.6.5 – Changes

Security

- SERVER-15465[141] OpenSSL crashes on stepdown
- SERVER-15360[142] User document changes made on a 2.4 primary and replicated to a 2.6 secondary don't make the 2.6 secondary invalidate its user cache
- SERVER-14887[143] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14727[144] Details of SASL failures aren't logged
- SERVER-12551[145] Audit DML/CRUD operations

Stability SERVER-9032[146] mongod fails when launched with misconfigured locale

[132] https://jira.mongodb.org/browse/SERVER-13471
[133] https://jira.mongodb.org/browse/SERVER-16324
[134] https://jira.mongodb.org/browse/SERVER-10069
[135] https://jira.mongodb.org/browse/SERVER-15632
[136] https://jira.mongodb.org/browse/SERVER-15629
[137] https://jira.mongodb.org/browse/SERVER-16316
[138] https://jira.mongodb.org/browse/SERVER-14763
[139] https://jira.mongodb.org/browse/SERVER-14249
[140] https://jira.mongodb.org/browse/SERVER-13726
[141] https://jira.mongodb.org/browse/SERVER-15465
[142] https://jira.mongodb.org/browse/SERVER-15360
[143] https://jira.mongodb.org/browse/SERVER-14887
[144] https://jira.mongodb.org/browse/SERVER-14727
[145] https://jira.mongodb.org/browse/SERVER-12551
[146] https://jira.mongodb.org/browse/SERVER-9032

Querying

- SERVER-15287[147] Query planner sort analysis incorrectly allows index key pattern plugin fields to provide sort
- SERVER-15286[148] Assertion in date indexes when opposite-direction-sorted and double "or" filtered
- SERVER-15279[149] Disable hash-based index intersection (AND_HASH) by default
- SERVER-15152[150] When evaluating plans, some index candidates cause complete index scan
- SERVER-15015[151] Assertion failure when combining `$max` and `$min` and reverse index scan
- SERVER-15012[152] Server crashes on indexed rooted $or queries using a 2d index
- SERVER-14969[153] Dropping index during active aggregation operation can crash server
- SERVER-14961[154] Plan ranker favors intersection plans if predicate generates empty range index scan
- SERVER-14892[155] Invalid `{$elemMatch: {$where}}` query causes memory leak
- SERVER-14706[156] Queries that use negated $type predicate over a field may return incomplete results when an index is present on that field
- SERVER-13104[157] Plan enumerator doesn't enumerate all possibilities for a nested `$or`
- SERVER-14984[158] Server aborts when running `$centerSphere` query with `NaN` radius
- SERVER-14981[159] Server aborts when querying against `2dsphere` index with `coarsestIndexedLevel:0`
- SERVER-14831[160] Text search trips assertion when default language only supported in `textIndexVersion=1` used

Replication

- SERVER-15038[161] Multiple background index builds may not interrupt cleanly for commands, on secondaries
- SERVER-14887[162] Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14805[163] Use multithreaded oplog replay during initial sync

Sharding

- SERVER-15056[164] Sharded connection cleanup on setup error can crash mongos
- SERVER-13702[165] Commands without optional query may target to wrong shards on mongos

[147] https://jira.mongodb.org/browse/SERVER-15287
[148] https://jira.mongodb.org/browse/SERVER-15286
[149] https://jira.mongodb.org/browse/SERVER-15279
[150] https://jira.mongodb.org/browse/SERVER-15152
[151] https://jira.mongodb.org/browse/SERVER-15015
[152] https://jira.mongodb.org/browse/SERVER-15012
[153] https://jira.mongodb.org/browse/SERVER-14969
[154] https://jira.mongodb.org/browse/SERVER-14961
[155] https://jira.mongodb.org/browse/SERVER-14892
[156] https://jira.mongodb.org/browse/SERVER-14706
[157] https://jira.mongodb.org/browse/SERVER-13104
[158] https://jira.mongodb.org/browse/SERVER-14984
[159] https://jira.mongodb.org/browse/SERVER-14981
[160] https://jira.mongodb.org/browse/SERVER-14831
[161] https://jira.mongodb.org/browse/SERVER-15038
[162] https://jira.mongodb.org/browse/SERVER-14887
[163] https://jira.mongodb.org/browse/SERVER-14805
[164] https://jira.mongodb.org/browse/SERVER-15056
[165] https://jira.mongodb.org/browse/SERVER-13702

- SERVER-15156[166] MongoDB upgrade 2.4 to 2.6 check returns error in `config.changelog collection`

Storage

- SERVER-15369[167] explicitly zero .ns files on creation
- SERVER-15319[168] Verify 2.8 freelist is upgrade-downgrade safe with 2.6
- SERVER-15111[169] partially written journal last section causes recovery to fail

Indexing

- SERVER-14848[170] Port `index_id_desc.js` to v2.6 and master branches
- SERVER-14205[171] ensureIndex failure reports `ok: 1` on some failures

Write Operations

- SERVER-15106[172] Incorrect nscanned and nscannedObjects for idhack updates in 2.6.4 profiler or slow query log
- SERVER-15029[173] The `$rename` modifier uses incorrect dotted source path
- SERVER-14829[174] `UpdateIndexData::clear()` should reset all member variables

Data Aggregation

- SERVER-15087[175] Server crashes when running concurrent mapReduce and dropDatabase commands
- SERVER-14969[176] Dropping index during active aggregation operation can crash server
- SERVER-14168[177] Warning logged when incremental MR collections are unsuccessfully dropped on secondaries

Packaging

- SERVER-14679[178] (CentOS 7/RHEL 7) `init.d` script should create directory for `pid` file if it is missing
- SERVER-14023[179] Support for RHEL 7 Enterprise `.rpm` packages
- SERVER-13243[180] Support for Ubuntu 14 "Trusty" Enterprise `.deb` packages
- SERVER-11077[181] Support for Debian 7 Enterprise `.deb` packages
- SERVER-10642[182] Generate Community and Enterprise packages for SUSE 11

[166] https://jira.mongodb.org/browse/SERVER-15156
[167] https://jira.mongodb.org/browse/SERVER-15369
[168] https://jira.mongodb.org/browse/SERVER-15319
[169] https://jira.mongodb.org/browse/SERVER-15111
[170] https://jira.mongodb.org/browse/SERVER-14848
[171] https://jira.mongodb.org/browse/SERVER-14205
[172] https://jira.mongodb.org/browse/SERVER-15106
[173] https://jira.mongodb.org/browse/SERVER-15029
[174] https://jira.mongodb.org/browse/SERVER-14829
[175] https://jira.mongodb.org/browse/SERVER-15087
[176] https://jira.mongodb.org/browse/SERVER-14969
[177] https://jira.mongodb.org/browse/SERVER-14168
[178] https://jira.mongodb.org/browse/SERVER-14679
[179] https://jira.mongodb.org/browse/SERVER-14023
[180] https://jira.mongodb.org/browse/SERVER-13243
[181] https://jira.mongodb.org/browse/SERVER-11077
[182] https://jira.mongodb.org/browse/SERVER-10642

Logging and Diagnostics

- SERVER-14964[183] nscanned not written to the logs at `logLevel` 1 unless `slowms` exceeded or profiling enabled
- SERVER-12551[184] Audit DML/CRUD operations
- SERVER-14904[185] Adjust dates in `tool/exportimport_date.js` to account for different timezones

Internal Code and Testing

- SERVER-13770[186] `Helpers::removeRange` should check all runner states
- SERVER-14284[187] jstests should not leave profiler enabled at test run end
- SERVER-14076[188] remove test `replset_remove_node.js`
- SERVER-14778[189] Hide function and data pointers for natively-injected v8 functions

2.6.4 – Changes

Security

- SERVER-14701[190] The "backup" auth role should allow running the "collstats" command for all resources
- SERVER-14518[191] Allow disabling hostname validation for SSL
- SERVER-14268[192] Potential information leak
- SERVER-14170[193] Cannot read from secondary if both audit and auth are enabled in a sharded cluster
- SERVER-13833[194] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles
- SERVER-12512[195] Add role-based, selective audit logging.
- SERVER-9482[196] Add build flag for sslFIPSMode

Querying

- SERVER-14625[197] Query planner can construct incorrect bounds for negations inside $elemMatch
- SERVER-14607[198] hash intersection of fetched and non-fetched data can discard data from a result
- SERVER-14532[199] Improve logging in the case of plan ranker ties

[183] https://jira.mongodb.org/browse/SERVER-14964
[184] https://jira.mongodb.org/browse/SERVER-12551
[185] https://jira.mongodb.org/browse/SERVER-14904
[186] https://jira.mongodb.org/browse/SERVER-13770
[187] https://jira.mongodb.org/browse/SERVER-14284
[188] https://jira.mongodb.org/browse/SERVER-14076
[189] https://jira.mongodb.org/browse/SERVER-14778
[190] https://jira.mongodb.org/browse/SERVER-14701
[191] https://jira.mongodb.org/browse/SERVER-14518
[192] https://jira.mongodb.org/browse/SERVER-14268
[193] https://jira.mongodb.org/browse/SERVER-14170
[194] https://jira.mongodb.org/browse/SERVER-13833
[195] https://jira.mongodb.org/browse/SERVER-12512
[196] https://jira.mongodb.org/browse/SERVER-9482
[197] https://jira.mongodb.org/browse/SERVER-14625
[198] https://jira.mongodb.org/browse/SERVER-14607
[199] https://jira.mongodb.org/browse/SERVER-14532

- SERVER-14350[200] Server crash when $centerSphere has non-positive radius
- SERVER-14317[201] Dead code in IDHackRunner::applyProjection
- SERVER-14311[202] skipping of index keys is not accounted for in plan ranking by the index scan stage
- SERVER-14123[203] some operations can create BSON object larger than the 16MB limit
- SERVER-14034[204] Sorted $in query with large number of elements can't use merge sort
- SERVER-13994[205] do not aggressively pre-fetch data for parallelCollectionScan

Replication

- SERVER-14665[206] Build failure for v2.6 in closeall.js caused by access violation reading _me
- SERVER-14505[207] cannot dropAllIndexes when index builds in progress assertion failure
- SERVER-14494[208] Dropping collection during active background index build on secondary triggers segfault
- SERVER-13822[209] Running resync before replset config is loaded can crash `mongod`
- SERVER-11776[210] Replication 'isself' check should allow mapped ports

Sharding

- SERVER-14551[211] Runner yield during migration cleanup (removeRange) results in fassert
- SERVER-14431[212] Invalid chunk data after splitting on a key that's too large
- SERVER-14261[213] stepdown during migration range delete can abort `mongod`
- SERVER-14032[214] v2.6 `mongos` doesn't verify _id is present for config server upserts
- SERVER-13648[215] better stats from migration cleanup
- SERVER-12750[216] `mongos` shouldn't accept initial query with "exhaust" flag set
- SERVER-9788[217] `mongos` does not re-evaluate read preference once a valid replica set member is chosen
- SERVER-9526[218] Log messages regarding chunks not very informative when the shard key is of type BinData

[200] https://jira.mongodb.org/browse/SERVER-14350
[201] https://jira.mongodb.org/browse/SERVER-14317
[202] https://jira.mongodb.org/browse/SERVER-14311
[203] https://jira.mongodb.org/browse/SERVER-14123
[204] https://jira.mongodb.org/browse/SERVER-14034
[205] https://jira.mongodb.org/browse/SERVER-13994
[206] https://jira.mongodb.org/browse/SERVER-14665
[207] https://jira.mongodb.org/browse/SERVER-14505
[208] https://jira.mongodb.org/browse/SERVER-14494
[209] https://jira.mongodb.org/browse/SERVER-13822
[210] https://jira.mongodb.org/browse/SERVER-11776
[211] https://jira.mongodb.org/browse/SERVER-14551
[212] https://jira.mongodb.org/browse/SERVER-14431
[213] https://jira.mongodb.org/browse/SERVER-14261
[214] https://jira.mongodb.org/browse/SERVER-14032
[215] https://jira.mongodb.org/browse/SERVER-13648
[216] https://jira.mongodb.org/browse/SERVER-12750
[217] https://jira.mongodb.org/browse/SERVER-9788
[218] https://jira.mongodb.org/browse/SERVER-9526

Storage

- SERVER-14198[219] Std::set<pointer> and Windows Heap Allocation Reuse produces non-deterministic results
- SERVER-13975[220] Creating index on collection named "system" can cause server to abort
- SERVER-13729[221] Reads & Writes are blocked during data file allocation on Windows
- SERVER-13681[222] `mongod` B stalls during background flush on Windows

Indexing SERVER-14494[223] Dropping collection during active background index build on secondary triggers segfault

Write Ops

- SERVER-14257[224] "remove" command can cause process termination by throwing unhandled exception if profiling is enabled
- SERVER-14024[225] Update fails when query contains part of a DBRef and results in an insert (upsert:true)
- SERVER-13764[226] debug mechanisms report incorrect nscanned / nscannedObjects for updates

Networking SERVER-13734[227] Remove catch (...) from handleIncomingMsg

Geo

- SERVER-14039[228] $nearSphere query with 2d index, skip, and limit returns incomplete results
- SERVER-13701[229] Query using 2d index throws exception when using explain()

Text Search

- SERVER-14738[230] Updates to documents with text-indexed fields may lead to incorrect entries
- SERVER-14027[231] Renaming collection within same database fails if wildcard text index present

Tools

- SERVER-14212[232] `mongorestore` may drop system users and roles
- SERVER-14048[233] `mongodump` against `mongos` can't send dump to standard output

[219] https://jira.mongodb.org/browse/SERVER-14198
[220] https://jira.mongodb.org/browse/SERVER-13975
[221] https://jira.mongodb.org/browse/SERVER-13729
[222] https://jira.mongodb.org/browse/SERVER-13681
[223] https://jira.mongodb.org/browse/SERVER-14494
[224] https://jira.mongodb.org/browse/SERVER-14257
[225] https://jira.mongodb.org/browse/SERVER-14024
[226] https://jira.mongodb.org/browse/SERVER-13764
[227] https://jira.mongodb.org/browse/SERVER-13734
[228] https://jira.mongodb.org/browse/SERVER-14039
[229] https://jira.mongodb.org/browse/SERVER-13701
[230] https://jira.mongodb.org/browse/SERVER-14738
[231] https://jira.mongodb.org/browse/SERVER-14027
[232] https://jira.mongodb.org/browse/SERVER-14212
[233] https://jira.mongodb.org/browse/SERVER-14048

Admin

- SERVER-14556[234] Default dbpath for `mongod --configsvr` changes in 2.6
- SERVER-14355[235] Allow dbAdmin role to manually create system.profile collections

Packaging SERVER-14283[236] Parameters in installed config file are out of date

JavaScript

- SERVER-14254[237] Do not store native function pointer as a property in function prototype
- SERVER-13798[238] v8 garbage collection can cause crash due to independent lifetime of DBClient and Cursor objects
- SERVER-13707[239] mongo shell may crash when converting invalid regular expression

Shell

- SERVER-14341[240] negative opcounter values in serverStatus
- SERVER-14107[241] Querying for a document containing a value of either type Javascript or JavascriptWithScope crashes the shell

Usability SERVER-13833[242] userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles

Logging and Diagnostics

- SERVER-12512[243] Add role-based, selective audit logging.
- SERVER-14341[244] negative opcounter values in serverStatus

Testing

- SERVER-14731[245] plan_cache_ties.js sometimes fails
- SERVER-14147[246] make index_multi.js retry on connection failure
- SERVER-13615[247] sharding_rs2.js intermittent failure due to reliance on opcounters

[234] https://jira.mongodb.org/browse/SERVER-14556
[235] https://jira.mongodb.org/browse/SERVER-14355
[236] https://jira.mongodb.org/browse/SERVER-14283
[237] https://jira.mongodb.org/browse/SERVER-14254
[238] https://jira.mongodb.org/browse/SERVER-13798
[239] https://jira.mongodb.org/browse/SERVER-13707
[240] https://jira.mongodb.org/browse/SERVER-14341
[241] https://jira.mongodb.org/browse/SERVER-14107
[242] https://jira.mongodb.org/browse/SERVER-13833
[243] https://jira.mongodb.org/browse/SERVER-12512
[244] https://jira.mongodb.org/browse/SERVER-14341
[245] https://jira.mongodb.org/browse/SERVER-14731
[246] https://jira.mongodb.org/browse/SERVER-14147
[247] https://jira.mongodb.org/browse/SERVER-13615

2.6.3 – Changes

- SERVER-14302[248] Fixed: "Equality queries on `_id` with projection may return no results on sharded collections"
- SERVER-14304[249] Fixed: "Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections"

2.6.2 – Changes

Security

- SERVER-13727[250] The `backup` (page 410) authorization role now includes privileges to run the `collStats` command.
- SERVER-13804[251] The built-in role `restore` (page 410) now has privileges on `system.roles` collection.
- SERVER-13612[252] Fixed: "SSL-enabled server appears not to be sending the list of supported certificate issuers to the client"
- SERVER-13753[253] Fixed: "`mongod` may terminate if x.509 authentication certificate is invalid"
- SERVER-13945[254] For *replica set/sharded cluster member authentication* (page 359), now matches x.509 cluster certificates by attributes instead of by substring comparison.
- SERVER-13868[255] Now marks V1 users as probed on databases that do not have surrogate user documents.
- SERVER-13850[256] Now ensures that the user cache entry is up to date before using it to determine a user's roles in user management commands on `mongos`.
- SERVER-13588[257] Fixed: "Shell prints startup warning when auth enabled"

Querying

- SERVER-13731[258] Fixed: "Stack overflow when parsing deeply nested `$not` query"
- SERVER-13890[259] Fixed: "Index bounds builder constructs invalid bounds for multiple negations joined by an `$or`"
- SERVER-13752[260] Verified assertion on empty `$in` clause and sort on second field in a compound index.
- SERVER-13337[261] Re-enabled `idhack` for queries with projection.
- SERVER-13715[262] Fixed: "Aggregation pipeline execution can fail with $or and blocking sorts"
- SERVER-13714[263] Fixed: "non-top-level indexable `$not` triggers query planning bug"

[248] https://jira.mongodb.org/browse/SERVER-14302
[249] https://jira.mongodb.org/browse/SERVER-14304
[250] https://jira.mongodb.org/browse/SERVER-13727
[251] https://jira.mongodb.org/browse/SERVER-13804
[252] https://jira.mongodb.org/browse/SERVER-13612
[253] https://jira.mongodb.org/browse/SERVER-13753
[254] https://jira.mongodb.org/browse/SERVER-13945
[255] https://jira.mongodb.org/browse/SERVER-13868
[256] https://jira.mongodb.org/browse/SERVER-13850
[257] https://jira.mongodb.org/browse/SERVER-13588
[258] https://jira.mongodb.org/browse/SERVER-13731
[259] https://jira.mongodb.org/browse/SERVER-13890
[260] https://jira.mongodb.org/browse/SERVER-13752
[261] https://jira.mongodb.org/browse/SERVER-13337
[262] https://jira.mongodb.org/browse/SERVER-13715
[263] https://jira.mongodb.org/browse/SERVER-13714

- SERVER-13769[264] Fixed: "`distinct` command on indexed field with geo predicate fails to execute"
- SERVER-13675[265] Fixed "Plans with differing performance can tie during plan ranking"
- SERVER-13899[266] Fixed: "'Whole index scan' query solutions can use incompatible indexes, return incorrect results"
- SERVER-13852[267] Fixed "IndexBounds::endKeyInclusive not initialized by constructor"
- SERVER-14073[268] planSummary no longer truncated at 255 characters
- SERVER-14174[269] Fixed: "If ntoreturn is a limit (rather than batch size) extra data gets buffered during plan ranking"
- SERVER-13789[270] Some nested queries no longer trigger an assertion error
- SERVER-14064[271] Added planSummary information for `count` command log message.
- SERVER-13960[272] Queries containing `$or` no longer miss results if multiple clauses use the same index.
- SERVER-14180[273] Fixed: "Crash with 'and' clause, `$elemMatch`, and nested `$mod` or regex"
- SERVER-14176[274] Natural order sort specification no longer ignored if query is specified.
- SERVER-13754[275] Bounds no longer combined for `$or` queries that can use merge sort.

Geospatial SERVER-13687[276] Results of `$near` query on compound multi-key 2dsphere index are now sorted by distance.

Write Operations SERVER-13802[277] Insert field validation no longer stops at first `Timestamp()` field.

Replication

- SERVER-13993[278] Fixed: "log a message when `shouldChangeSyncTarget()` believes a node should change sync targets"
- SERVER-13976[279] Fixed: "Cloner needs to detect failure to create collection"

Sharding

- SERVER-13616[280] Resolved: "'type 7' (OID) error when acquiring distributed lock for first time"
- SERVER-13812[281] Now catches exception thrown by `getShardsForQuery` for geo query.

[264] https://jira.mongodb.org/browse/SERVER-13769
[265] https://jira.mongodb.org/browse/SERVER-13675
[266] https://jira.mongodb.org/browse/SERVER-13899
[267] https://jira.mongodb.org/browse/SERVER-13852
[268] https://jira.mongodb.org/browse/SERVER-14073
[269] https://jira.mongodb.org/browse/SERVER-14174
[270] https://jira.mongodb.org/browse/SERVER-13789
[271] https://jira.mongodb.org/browse/SERVER-14064
[272] https://jira.mongodb.org/browse/SERVER-13960
[273] https://jira.mongodb.org/browse/SERVER-14180
[274] https://jira.mongodb.org/browse/SERVER-14176
[275] https://jira.mongodb.org/browse/SERVER-13754
[276] https://jira.mongodb.org/browse/SERVER-13687
[277] https://jira.mongodb.org/browse/SERVER-13802
[278] https://jira.mongodb.org/browse/SERVER-13993
[279] https://jira.mongodb.org/browse/SERVER-13976
[280] https://jira.mongodb.org/browse/SERVER-13616
[281] https://jira.mongodb.org/browse/SERVER-13812

- SERVER-14138[282] `mongos` will now correctly target multiple shards for nested field shard key predicates.
- SERVER-11332[283] Fixed: "Authentication requests delayed if first config server is unresponsive"

Map/Reduce

- SERVER-14186[284] Resolved: "`rs.stepDown` during mapReduce causes fassert in logOp"
- SERVER-13981[285] Temporary map/reduce collections are now correctly replicated to secondaries.

Storage

- SERVER-13750[286] `convertToCapped` on empty collection no longer aborts after `invariant()` failure.
- SERVER-14056[287] Moving large collection across databases with renameCollection no longer triggers fatal assertion.
- SERVER-14082[288] Fixed: "Excessive freelist scanning for MaxBucket"
- SERVER-13737[289] CollectionOptions parser now skips non-numeric for "size"/"max" elements if values non-numeric.

Build and Packaging

- SERVER-13950[290] MongoDB Enterprise now includes required dependency list.
- SERVER-13862[291] Support for mongodb-org-server installation 2.6.1-1 on RHEL5 via RPM.
- SERVER-13724[292] Added SCons flag to override treating all warnings as errors.

Diagnostics

- SERVER-13587[293] Resolved: "`ndeleted` (page 307) in `system.profile` documents reports 1 too few documents removed"
- SERVER-13368[294] Improved exposure of timing information in `currentOp`.

Administration SERVER-13954[295] `security.javascriptEnabled` option is now available in the YAML configuration file.

[282] https://jira.mongodb.org/browse/SERVER-14138
[283] https://jira.mongodb.org/browse/SERVER-11332
[284] https://jira.mongodb.org/browse/SERVER-14186
[285] https://jira.mongodb.org/browse/SERVER-13981
[286] https://jira.mongodb.org/browse/SERVER-13750
[287] https://jira.mongodb.org/browse/SERVER-14056
[288] https://jira.mongodb.org/browse/SERVER-14082
[289] https://jira.mongodb.org/browse/SERVER-13737
[290] https://jira.mongodb.org/browse/SERVER-13950
[291] https://jira.mongodb.org/browse/SERVER-13862
[292] https://jira.mongodb.org/browse/SERVER-13724
[293] https://jira.mongodb.org/browse/SERVER-13587
[294] https://jira.mongodb.org/browse/SERVER-13368
[295] https://jira.mongodb.org/browse/SERVER-13954

Tools

- SERVER-10464[296] `mongodump` can now query `oplog.$main` and `oplog.rs` when using *`--dbpath`*.
- SERVER-13760[297] `mongoexport` can now handle large timestamps on Windows.

Shell

- SERVER-13865[298] Shell now returns correct `WriteResult` for compatibility-mode upsert with non-OID equality predicate on `_id` field.
- SERVER-13037[299] Fixed typo in error message for "compatibility mode".

Internal Code

- SERVER-13794[300] Fixed: "Unused snapshot history consuming significant heap space"
- SERVER-13446[301] Removed Solaris builds dependency on ILLUMOS libc.
- SERVER-14092[302] MongoDB upgrade 2.4 to 2.6 check no longer returns an error in internal collections.
- SERVER-14000[303] Added new lsb file location for Debian 7.1

Testing

- SERVER-13723[304] Stabilized `tags.js` after a change in its timeout when it was ported to use write commands.
- SERVER-13494[305] Fixed: "`setup_multiversion_mongodb.py` doesn't download 2.4.10 because of non-numeric version sorting"
- SERVER-13603[306] Fixed: "Test suites with options tests fail when run with *`--noprealloc`*"
- SERVER-13948[307] Fixed: "`awaitReplication()` failures related to getting a config version from master causing test failures"
- SERVER-13839[308] Fixed `sync2.js` failure.
- SERVER-13972[309] Fixed `connections_opened.js` failure.
- SERVER-13712[310] Reduced peak disk usage of test suites.
- SERVER-14249[311] Added tests for querying oplog via `mongodump` using *`--dbpath`*
- SERVER-10462[312] Fixed: "Windows file locking related buildbot failures"

[296] https://jira.mongodb.org/browse/SERVER-10464
[297] https://jira.mongodb.org/browse/SERVER-13760
[298] https://jira.mongodb.org/browse/SERVER-13865
[299] https://jira.mongodb.org/browse/SERVER-13037
[300] https://jira.mongodb.org/browse/SERVER-13794
[301] https://jira.mongodb.org/browse/SERVER-13446
[302] https://jira.mongodb.org/browse/SERVER-14092
[303] https://jira.mongodb.org/browse/SERVER-14000
[304] https://jira.mongodb.org/browse/SERVER-13723
[305] https://jira.mongodb.org/browse/SERVER-13494
[306] https://jira.mongodb.org/browse/SERVER-13603
[307] https://jira.mongodb.org/browse/SERVER-13948
[308] https://jira.mongodb.org/browse/SERVER-13839
[309] https://jira.mongodb.org/browse/SERVER-13972
[310] https://jira.mongodb.org/browse/SERVER-13712
[311] https://jira.mongodb.org/browse/SERVER-14249
[312] https://jira.mongodb.org/browse/SERVER-10462

2.6.1 – Changes

Stability SERVER-13739[313] Repair database failure can delete database files

Build and Packaging

- SERVER-13287[314] Addition of debug symbols has doubled compile time
- SERVER-13563[315] Upgrading from 2.4.x to 2.6.0 via `yum` clobbers configuration file
- SERVER-13691[316] yum and apt "stable" repositories contain release candidate 2.6.1-rc0 packages
- SERVER-13515[317] Cannot install MongoDB as a service on Windows

Querying

- SERVER-13066[318] Negations over multikey fields do not use index
- SERVER-13495[319] Concurrent `GETMORE` and `KILLCURSORS` operations can cause race condition and server crash
- SERVER-13503[320] The `$where` operator should not be allowed under `$elemMatch`
- SERVER-13537[321] Large skip and and limit values can cause crash in blocking sort stage
- SERVER-13557[322] Incorrect negation of $elemMatch value in 2.6
- SERVER-13562[323] Queries that use tailable cursors do not stream results if skip() is applied
- SERVER-13566[324] Using the OplogReplay flag with extra predicates can yield incorrect results
- SERVER-13611[325] Missing sort order for compound index leads to unnecessary in-memory sort
- SERVER-13618[326] Optimization for sorted $in queries not applied to reverse sort order
- SERVER-13661[327] Increase the maximum allowed depth of query objects
- SERVER-13664[328] Query with `$elemMatch` using a compound multikey index can generate incorrect results
- SERVER-13677[329] Query planner should traverse through $all while handling $elemMatch object predicates
- SERVER-13766[330] Dropping index or collection while $or query is yielding triggers fatal assertion

[313] https://jira.mongodb.org/browse/SERVER-13739
[314] https://jira.mongodb.org/browse/SERVER-13287
[315] https://jira.mongodb.org/browse/SERVER-13563
[316] https://jira.mongodb.org/browse/SERVER-13691
[317] https://jira.mongodb.org/browse/SERVER-13515
[318] https://jira.mongodb.org/browse/SERVER-13066
[319] https://jira.mongodb.org/browse/SERVER-13495
[320] https://jira.mongodb.org/browse/SERVER-13503
[321] https://jira.mongodb.org/browse/SERVER-13537
[322] https://jira.mongodb.org/browse/SERVER-13557
[323] https://jira.mongodb.org/browse/SERVER-13562
[324] https://jira.mongodb.org/browse/SERVER-13566
[325] https://jira.mongodb.org/browse/SERVER-13611
[326] https://jira.mongodb.org/browse/SERVER-13618
[327] https://jira.mongodb.org/browse/SERVER-13661
[328] https://jira.mongodb.org/browse/SERVER-13664
[329] https://jira.mongodb.org/browse/SERVER-13677
[330] https://jira.mongodb.org/browse/SERVER-13766

Geospatial

- SERVER-13666[331] `$near` queries with out-of-bounds points in legacy format can lead to crashes
- SERVER-13540[332] The `geoNear` command no longer returns distance in radians for legacy point
- SERVER-13486[333]: The `geoNear` command can create too large BSON objects for aggregation.

Replication

- SERVER-13500[334] Changing replica set configuration can crash running members
- SERVER-13589[335] Background index builds from a 2.6.0 primary fail to complete on 2.4.x secondaries
- SERVER-13620[336] Replicated data definition commands will fail on secondaries during background index build
- SERVER-13496[337] Creating index with same name but different spec in mixed version replicaset can abort replication

Sharding

- SERVER-12638[338] Initial sharding with hashed shard key can result in duplicate split points
- SERVER-13518[339] The `_id` field is no longer automatically generated by `mongos` when missing
- SERVER-13777[340] Migrated ranges waiting for deletion do not report cursors still open

Security

- SERVER-9358[341] Log rotation can overwrite previous log files
- SERVER-13644[342] Sensitive credentials in startup options are not redacted and may be exposed
- SERVER-13441[343] Inconsistent error handling in user management shell helpers

Write Operations

- SERVER-13466[344] Error message in collection creation failure contains incorrect namespace
- SERVER-13499[345] Yield policy for batch-inserts should be the same as for batch-updates/deletes
- SERVER-13516[346] Array updates on documents with more than 128 BSON elements may crash `mongod`

[331] https://jira.mongodb.org/browse/SERVER-13666
[332] https://jira.mongodb.org/browse/SERVER-13540
[333] https://jira.mongodb.org/browse/SERVER-13486
[334] https://jira.mongodb.org/browse/SERVER-13500
[335] https://jira.mongodb.org/browse/SERVER-13589
[336] https://jira.mongodb.org/browse/SERVER-13620
[337] https://jira.mongodb.org/browse/SERVER-13496
[338] https://jira.mongodb.org/browse/SERVER-12638
[339] https://jira.mongodb.org/browse/SERVER-13518
[340] https://jira.mongodb.org/browse/SERVER-13777
[341] https://jira.mongodb.org/browse/SERVER-9358
[342] https://jira.mongodb.org/browse/SERVER-13644
[343] https://jira.mongodb.org/browse/SERVER-13441
[344] https://jira.mongodb.org/browse/SERVER-13466
[345] https://jira.mongodb.org/browse/SERVER-13499
[346] https://jira.mongodb.org/browse/SERVER-13516

2.6.11 – Aug 12, 2015

- Improvements to query plan ranking SERVER-17815[347]
- Improved ability for `mongos` to detect replica set failover and correctly route read operations to the new primary SERVER-18280[348]
- Improved reporting of queries in `getMore` operation in `db.currentOp()` and the database profiler SERVER-16265[349]
- All issues closed in 2.6.11[350]

2.6.10 – May 19, 2015

- Improve user cache invalidation enforcement on `mongos` SERVER-11980[351]
- Provide correct rollbacks for collection creation SERVER-18211[352]
- Allow user inserts into the `system.profile` collection SERVER-18211[353]
- Fix to query system to ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction SERVER-18364[354]
- All issues closed in 2.6.10[355]

2.6.9 – March 24, 2015

- Resolve connection handling related crash with `mongos` instances SERVER-17441[356]
- Add server parameter to configure idle cursor timeout SERVER-8188[357]
- Remove duplicated (orphan) documents from aggregation pipelines with `_id` queries in sharded clusters SERVER-17426[358]
- Fixed crash in `geoNear` queries with multiple `2dsphere` indexes SERVER-14723[359]
- All issues closed in 2.6.9[360]

2.6.8 – February 25, 2015

- Add `listCollections` command functionality to 2.6 shell and client SERVER-17087[361]
- `copydb`/`clone` commands can crash the server if a primary steps down SERVER-16599[362]

[347] https://jira.mongodb.org/browse/SERVER-17815
[348] https://jira.mongodb.org/browse/SERVER-18280
[349] https://jira.mongodb.org/browse/SERVER-16265
[350] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.11%22%20AND%20project%20%3D%20SERVER
[351] https://jira.mongodb.org/browse/SERVER-11980
[352] https://jira.mongodb.org/browse/SERVER-18211
[353] https://jira.mongodb.org/browse/SERVER-18211
[354] https://jira.mongodb.org/browse/SERVER-18364
[355] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.10%22%20AND%20project%20%3D%20SERVER
[356] https://jira.mongodb.org/browse/SERVER-17441
[357] https://jira.mongodb.org/browse/SERVER-8188
[358] https://jira.mongodb.org/browse/SERVER-17426
[359] https://jira.mongodb.org/browse/SERVER-14723
[360] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.9%22%20AND%20project%20%3D%20SERVER
[361] https://jira.mongodb.org/browse/SERVER-17087
[362] https://jira.mongodb.org/browse/SERVER-16599

- Secondary fasserts trying to replicate an index SERVER-16274[363]
- Query optimizer should always use equality predicate over unique index when possible SERVER-15802[364]
- All issues closed in 2.6.8[365]

2.6.7 – January 13, 2015

- Decreased `mongos` memory footprint when shards have several tags SERVER-16683[366]
- Removed check for shard version if the primary server is down SERVER-16237[367]
- Fixed: `/etc/init.d/mongod` startup script failure with dirname message SERVER-16081[368]
- Fixed: `mongos` can cause shards to hit the in-memory sort limit by requesting more results than needed SERVER-14306[369]
- All issues closed in 2.6.7[370]

2.6.6 – December 09, 2014

- Fixed: Evaluating candidate query plans with concurrent writes on same collection may crash `mongod` SERVER-15580[371]
- Fixed: 2.6 `mongod` crashes with segfault when added to a 2.8 replica set with 12 or more members SERVER-16107[372]
- Fixed: `$regex`, `$in` and `$sort` with index returns too many results SERVER-15696[373]
- Change: `moveChunk` will fail if there is data on the target shard and a required index does not exist. SERVER-12472[374]
- Primary should abort if encountered problems writing to the oplog SERVER-12058[375]
- All issues closed in 2.6.6[376]

2.6.5 – October 07, 2014

- `$rename` now uses correct dotted source paths SERVER-15029[377]
- Partially written journal last section does not affect recovery SERVER-15111[378]
- Explicitly zero `.ns` files on creation SERVER-15369[379]

[363] https://jira.mongodb.org/browse/SERVER-16274
[364] https://jira.mongodb.org/browse/SERVER-15802
[365] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.8%22%20AND%20project%20%3D%20SERVER
[366] https://jira.mongodb.org/browse/SERVER-16683
[367] https://jira.mongodb.org/browse/SERVER-16237
[368] https://jira.mongodb.org/browse/SERVER-16081
[369] https://jira.mongodb.org/browse/SERVER-14306
[370] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.7%22%20AND%20project%20%3D%20SERVER
[371] https://jira.mongodb.org/browse/SERVER-15580
[372] https://jira.mongodb.org/browse/SERVER-16107
[373] https://jira.mongodb.org/browse/SERVER-15696
[374] https://jira.mongodb.org/browse/SERVER-12472
[375] https://jira.mongodb.org/browse/SERVER-12058
[376] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.6%22%20AND%20project%20%3D%20SERVER
[377] https://jira.mongodb.org/browse/SERVER-15029
[378] https://jira.mongodb.org/browse/SERVER-15111
[379] https://jira.mongodb.org/browse/SERVER-15369

- Plan ranker will no longer favor intersection plans if predicate generates empty range index scan SERVER-14961[380]
- Generate Community and Enterprise packages for SUSE 11 SERVER-10642[381]
- All issues closed in 2.6.5[382]

2.6.4 – August 11, 2014

- Fix for `text` index where under specific circumstances, in-place updates to a `text`-indexed field may result in incorrect/incomplete results SERVER-14738[383]
- Check the size of the split point before performing a manual split chunk operation SERVER-14431[384]
- Ensure read preferences are re-evaluated by drawing secondary connections from a global pool and releasing back to the pool at the end of a query/command SERVER-9788[385]
- Allow read from secondaries when both audit and authorization are enabled in a sharded cluster SERVER-14170[386]
- All issues closed in 2.6.4[387]

2.6.3 – June 19, 2014

- Equality queries on `_id` with projection may return no results on sharded collections SERVER-14302[388].
- Equality queries on `_id` with projection on `_id` may return orphan documents on sharded collections SERVER-14304[389].
- All issues closed in 2.6.3[390].

2.6.2 – June 16, 2014

- Query plans with differing performance can tie during plan ranking SERVER-13675[391].
- `mongod` may terminate if x.509 authentication certificate is invalid SERVER-13753[392].
- Temporary map/reduce collections are incorrectly replicated to secondaries SERVER-13981[393].
- `mongos` incorrectly targets multiple shards for nested field shard key predicates SERVER-14138[394].
- `rs.stepDown()` during mapReduce causes `fassert` when writing to op log SERVER-14186[395].

[380] https://jira.mongodb.org/browse/SERVER-14961
[381] https://jira.mongodb.org/browse/SERVER-10642
[382] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.5%22%20AND%20project%20%3D%20SERVER
[383] https://jira.mongodb.org/browse/SERVER-14738
[384] https://jira.mongodb.org/browse/SERVER-14431
[385] https://jira.mongodb.org/browse/SERVER-9788
[386] https://jira.mongodb.org/browse/SERVER-14170
[387] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.4%22%20AND%20project%20%3D%20SERVER
[388] https://jira.mongodb.org/browse/SERVER-14302
[389] https://jira.mongodb.org/browse/SERVER-14304
[390] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.3%22%20AND%20project%20%3D%20SERVER
[391] https://jira.mongodb.org/browse/SERVER-13675
[392] https://jira.mongodb.org/browse/SERVER-13753
[393] https://jira.mongodb.org/browse/SERVER-13981
[394] https://jira.mongodb.org/browse/SERVER-14138
[395] https://jira.mongodb.org/browse/SERVER-14186

- All issues closed in 2.6.2[396].

2.6.1 – May 5, 2014

- Fix to install MongoDB service on Windows with the `--install` option SERVER-13515[397].
- Allow direct upgrade from 2.4.x to 2.6.0 via `yum` SERVER-13563[398].
- Fix issues with background index builds on secondaries: SERVER-13589[399] and SERVER-13620[400].
- Redact credential information passed as startup options SERVER-13644[401].
- *2.6.1 Changelog* (page 826).
- All issues closed in 2.6.1[402].

Major Changes

The following changes in MongoDB affect both the standard and Enterprise editions:

Aggregation Enhancements

The aggregation pipeline adds the ability to return result sets of any size, either by returning a cursor or writing the output to a collection. Additionally, the aggregation pipeline supports variables and adds new operations to handle sets and redact data.

- The `db.collection.aggregate()` now returns a cursor, which enables the aggregation pipeline to return result sets of any size.
- Aggregation pipelines now support an `explain` operation to aid analysis of aggregation operations.
- Aggregation can now use a more efficient external-disk-based sorting process.
- New pipeline stages:
  - `$out` stage to output to a collection.
  - `$redact` stage to allow additional control to accessing the data.
- New or modified operators:
  - `set expression operators`.
  - `$let` and `$map` operators to allow for the use of variables.
  - `$literal` operator and `$size` operator.
  - `$cond` expression now accepts either an object or an array.

[396] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.2%22%20AND%20project%20%3D%20SERVER
[397] https://jira.mongodb.org/browse/SERVER-13515
[398] https://jira.mongodb.org/browse/SERVER-13563
[399] https://jira.mongodb.org/browse/SERVER-13589
[400] https://jira.mongodb.org/browse/SERVER-13620
[401] https://jira.mongodb.org/browse/SERVER-13644
[402] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.1%22%20AND%20project%20%3D%20SERVER

Text Search Integration

Text search is now enabled by default, and the query system, including the aggregation pipeline `$match` stage, includes the `$text` operator, which resolves text-search queries.

MongoDB 2.6 includes an updated *text index* (page 501) format and deprecates the `text` command.

Insert and Update Improvements

Improvements to the update and insert systems include additional operations and improvements that increase consistency of modified data.

- MongoDB preserves the order of the document fields following write operations *except* for the following cases:
  - The `_id` field is always the first field in the document.
  - Updates that include `renaming` of field names may result in the reordering of fields in the document.
- New or enhanced update operators:
  - `$bit` operator supports bitwise `xor` operation.
  - `$min` and `$max` operators that perform conditional update depending on the relative size of the specified value and the current value of a field.
  - `$push` operator has enhanced support for the `$sort`, `$slice`, and `$each` modifiers and supports a new `$position` modifier.
  - `$currentDate` operator to set the value of a field to the current date.
- The `$mul` operator for multiplicative increments for insert and update operations.

See also:

Update Operator Syntax Validation (page 842)

New Write Operation Protocol

A new write protocol integrates write operations with write concerns. The protocol also provides improved support for bulk operations.

MongoDB 2.6 adds the write commands `insert`, `update`, and `delete`, which provide the basis for the improved bulk insert. All officially supported MongoDB drivers support the new write commands.

The `mongo` shell now includes methods to perform bulk-write operations. See `Bulk()` for more information.

See also:

Write Method Acknowledgements (page 838)

MSI Package for MongoDB Available for Windows

MongoDB now distributes MSI packages for Microsoft Windows. This is the recommended method for MongoDB installation under Windows.

Security Improvements

MongoDB 2.6 enhances support for secure deployments through improved TLS/SSL support, x.509-based authentication, an improved authorization system with more granular controls, as well as centralized credential storage, and improved user management tools.

Specifically these changes include:

- A new *authorization model* (page 320) that provides the ability to create custom *User-Defined Roles* (page 321) and the ability to specify user privileges at a collection-level granularity.
- Global user management, which stores all user and user-defined role data in the `admin` database and provides a new set of commands for managing users and roles.
- x.509 certificate authentication for *client authentication* (page 357) as well as for *internal authentication* (page 359) of sharded and/or replica set cluster members. x.509 authentication is only available for deployments using TLS/SSL.
- Enhanced TLS/SSL Support:
  - *Rolling upgrades of clusters* (page 346) to use TLS/SSL.
  - *MongoDB Tools* (page 345) support connections to `mongod` and `mongos` instances using TLS/SSL connections.
  - *Prompt for passphrase* (page 341) by `mongod` or `mongos` at startup.
  - Require the use of strong TLS/SSL ciphers, with a minimum 128-bit key length for all connections. The strong-cipher requirement prevents an old or malicious client from forcing use of a weak cipher.
- MongoDB disables the http interface by default, limiting *network exposure* (page 322). To enable the interface, see `enabled`.

See also:

New Authorization Model (page 840), *TLS/SSL Certificate Hostname Validation* (page 840), and *Security Checklist* (page 431).

Query Engine Improvements

- MongoDB can now use *index intersection* (page 512) to fulfill queries supported by more than one index.
- *Index Filters* (page 73) to limit which indexes can become the winning plan for a query.
- `http://docs.mongodb.org/manual/reference/method/js-plan-cache` methods to view and clear the *query plans* (page 72) cached by the query optimizer.
- MongoDB can now use `count()` with `hint()`. See `count()` for details.

Improvements

Geospatial Enhancements

- *2dsphere indexes version 2* (page 497).
- Support for *MultiPoint* (page 560), *MultiLineString* (page 560), *MultiPolygon* (page 561), and *GeometryCollection* (page 561).
- Support for geospatial query clauses in `$or` expressions.

See also:

2dsphere Index Version 2 (page 841), *$maxDistance Changes* (page 843), *Deprecated $uniqueDocs* (page 844), *Stronger Validation of Geospatial Queries* (page 844)

Index Build Enhancements

- *Background index build* (page 511) allowed on secondaries. If you initiate a background index build on a *primary*, the secondaries will replicate the index build in the background.
- Automatic rebuild of interrupted index builds after a restart.
  - If a standalone or a primary instance terminates during an index build *without a clean shutdown*, `mongod` now restarts the index build when the instance restarts. If the instance shuts down cleanly or if a user kills the index build, the interrupted index builds do not automatically restart upon the restart of the server.
  - If a secondary instance terminates during an index build, the `mongod` instance will now restart the interrupted index build when the instance restarts.

  To disable this behavior, use the `--noIndexBuildRetry` command-line option.
- `ensureIndex()` now wraps a new `createIndex` command.
- The `dropDups` option to `ensureIndex()` and `createIndex` is deprecated.

See also:

Enforce Index Key Length Limit (page 837)

Enhanced Sharding and Replication Administration

- New `cleanupOrphaned` command to remove *orphaned documents* from a shard.
- New `mergeChunks` command to combine contiguous chunks located on a single shard. See `mergeChunks` and *Merge Chunks in a Sharded Cluster* (page 740).
- New `rs.printReplicationInfo()` and `rs.printSlaveReplicationInfo()` methods to provide a formatted report of the status of a replica set from the perspective of the primary and the secondary, respectively.

Configuration Options YAML File Format

MongoDB 2.6 supports a YAML-based configuration file format in addition to the previous configuration file format. See the documentation of the `Configuration File` for more information.

Operational Changes

Storage

`usePowerOf2Sizes` is now the default allocation strategy for all new collections. The new allocation strategy uses more storage relative to total document size but results in lower levels of storage fragmentation and more predictable storage capacity planning over time.

To use the previous *exact-fit allocation strategy*:

- For a specific collection, use `collMod` with `usePowerOf2Sizes` set to `false`.

- For all new collections on an entire `mongod` instance, set `newCollectionsUsePowerOf2Sizes` to `false`.

  New collections include those: created during *initial sync* (page 598), as well as those created by the `mongorestore` and `mongoimport` tools, by running `mongod` with the `--repair` option, as well as the `restoreDatabase` command.

See *Storage* (page 94) for more information about MongoDB's storage system.

Networking

- Removed upward limit for the `maxIncomingConnections` for `mongod` and `mongos`. Previous versions capped the maximum possible `maxIncomingConnections` setting at `20,000` connections.
- Connection pools for a `mongos` instance may be used by multiple MongoDB servers. This can reduce the number of connections needed for high-volume workloads and reduce resource consumption in sharded clusters.
- The C++ driver now monitors *replica set* health with the `isMaster` command instead of `replSetGetStatus`. This allows the C++ driver to support systems that require authentication.
- New `cursor.maxTimeMS()` and corresponding `maxTimeMS` option for commands to specify a time limit.

Tool Improvements

- `mongo` shell supports a global */etc/mongorc.js*.
- All MongoDB `executable files` now support the `--quiet` option to suppress all logging output except for error messages.
- `mongoimport` uses the input filename, without the file extension if any, as the collection name if run without the `-c` or `--collection` specification.
- `mongoexport` can now constrain export data using `--skip` and `--limit`, as well as order the documents in an export using the `--sort` option.
- `mongostat` can support the use of `--rowcount` (`-n`) with the `--discover` option to produce the specified number of output lines.
- Add strict mode representation for `data_numberlong` for use by `mongoexport` and `mongoimport`.

MongoDB Enterprise Features

The following changes are specific to MongoDB Enterprise Editions:

MongoDB Enterprise for Windows

MongoDB Enterprise for Windows (page 44) is now available. It includes support for Kerberos, SSL, and SNMP.

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB Enterprise for Windows includes OpenSSL version 1.0.1g.

Auditing

MongoDB Enterprise adds *auditing* (page 325) capability for `mongod` and `mongos` instances. See *Auditing* (page 325) for details.

LDAP Support for Authentication

MongoDB Enterprise provides support for proxy authentication of users. This allows administrators to configure a MongoDB cluster to authenticate users by proxying authentication requests to a specified Lightweight Directory Access Protocol (LDAP) service. See *Authenticate Using SASL and LDAP with OpenLDAP* (page 366) and *Authenticate Using SASL and LDAP with ActiveDirectory* (page 363) for details.

MongoDB Enterprise for Windows does **not** include LDAP support for authentication. However, MongoDB Enterprise for Linux supports using LDAP authentication with an ActiveDirectory server.

MongoDB does **not** support LDAP authentication in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 847) for upgrade instructions.

Expanded SNMP Support

MongoDB Enterprise has greatly expanded its SNMP support to provide SNMP access to nearly the full range of metrics provided by `db.serverStatus()`.

See also:

SNMP Changes (page 841)

Additional Information

Changes Affecting Compatibility

Compatibility Changes in MongoDB 2.6

On this page

The following 2.6 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 2.6* (page 805) for the full list of the 2.6 changes.

Index Changes

Enforce Index Key Length Limit

Description MongoDB 2.6 implements a stronger enforcement of the limit on `index key`.

Creating indexes will error if an index key in an existing document exceeds the limit:

- `db.collection.ensureIndex()`, `db.collection.reIndex()`, `compact`, and `repairDatabase` will error and not create the index. Previous versions of MongoDB would create the index but not index such documents.
- Because `db.collection.reIndex()`, `compact`, and `repairDatabase` drop *all* the indexes from a collection and then recreate them sequentially, the error from the index key limit prevents these operations from rebuilding any remaining indexes for the collection and, in the case of the `repairDatabase` command, from continuing with the remainder of the process.

Inserts will error:

- `db.collection.insert()` and other operations that perform inserts (e.g. `db.collection.save()` and `db.collection.update()` with `upsert` that result in inserts) will fail to insert if the new document has an indexed field whose corresponding index entry exceeds the limit. Previous versions of MongoDB would insert but not index such documents.
- `mongorestore` and `mongoimport` will fail to insert if the new document has an indexed field whose corresponding index entry exceeds the limit.

Updates will error:

- `db.collection.update()` and `db.collection.save()` operations on an indexed field will error if the updated value causes the index entry to exceed the limit.
- If an existing document contains an indexed field whose index entry exceeds the limit, updates on other fields that result in the relocation of a document on disk will error.

Chunk Migration will fail:

- Migrations will fail for a chunk that has a document with an indexed field whose index entry exceeds the limit.
- If left unfixed, the chunk will repeatedly fail migration, effectively ceasing chunk balancing for that collection. Or, if chunk splits occur in response to the migration failures, this response would lead to unnecessarily large number of chunks and an overly large config databases.

Secondary members of replica sets will warn:

- Secondaries will continue to replicate documents with an indexed field whose corresponding index entry exceeds the limit on initial sync but will print warnings in the logs.
- Secondaries allow index build and rebuild operations on a collection that contains an indexed field whose corresponding index entry exceeds the limit but with warnings in the logs.
- With *mixed version* replica sets where the secondaries are version 2.6 and the primary is version 2.4, secondaries will replicate documents inserted or updated on the 2.4 primary, but will print error messages in the log if the documents contain an indexed field whose corresponding index entry exceeds the limit.

Solution Run `db.upgradeCheckAllDBs()` to find current keys that violate this limit and correct as appropriate. Preferably, run the test before upgrading; i.e. connect the 2.6 `mongo` shell to your MongoDB 2.4 database and run the method.

If you have an existing data set and want to disable the default index key length validation so that you can upgrade before resolving these indexing issues, use the `failIndexKeyTooLong` parameter.

Index Specifications Validate Field Names

Description In MongoDB 2.6, create and re-index operations fail when the index key refers to an empty field, e.g. `"a..b" : 1` or the field name starts with a dollar sign (`$`).

- `db.collection.ensureIndex()` will not create a new index with an invalid or empty key name.
- `db.collection.reIndex()`, `compact`, and `repairDatabase` will error if an index exists with an invalid or empty key name.
- Chunk migration will fail if an index exists with an invalid or empty key name.

Previous versions of MongoDB allow the index.

Solution Run `db.upgradeCheckAllDBs()` to find current keys that violate this limit and correct as appropriate. Preferably, run the test before upgrading; i.e. connect the 2.6 `mongo` shell to your MongoDB 2.4 database and run the method.

`ensureIndex` and Existing Indexes

Description `db.collection.ensureIndex()` now errors:

- if you try to create an existing index but with different options; e.g. in the following example, the second `db.collection.ensureIndex()` will error.

```
db.mycollection.ensureIndex( { x: 1 } )
db.mycollection.ensureIndex( { x: 1 }, { unique: 1 } )
```

- if you specify an index name that already exists but the key specifications differ; e.g. in the following example, the second `db.collection.ensureIndex()` will error.

```
db.mycollection.ensureIndex( { a: 1 }, { name: "myIdx" } )
db.mycollection.ensureIndex( { z: 1 }, { name: "myIdx" } )
```

Previous versions did not create the index but did not error.

Write Method Acknowledgements

Description The `mongo` shell write methods `db.collection.insert()`, `db.collection.update()`, `db.collection.save()` and `db.collection.remove()` now integrate the *write concern* (page 82) directly into the method rather than with a separate `getLastError` command to provide *safe writes* (page 82) whether run interactively in the `mongo` shell or non-interactively in a script. In previous versions, these methods exhibited a "fire-and-forget" behavior. [403]

- Existing scripts for the `mongo` shell that used these methods will now observe safe writes which take **longer** than the previous "fire-and-forget" behavior.
- The write methods now return a `WriteResult` object that contains the results of the operation, including any write errors and write concern errors, and obviates the need to call `getLastError` command to get the status of the results. See `db.collection.insert()`, `db.collection.update()`, `db.collection.save()` and `db.collection.remove()` for details.
- In sharded environments, `mongos` no longer supports "fire-and-forget" behavior. This limits throughput when writing data to sharded clusters.

[403] In previous versions, when using the `mongo` shell interactively, the `mongo` shell automatically called the `getLastError` command after a write method to provide "safe writes". Scripts, however, would observe "fire-and-forget" behavior in previous versions unless the scripts included an **explicit** call to the `getLastError` command after a write method.

Solution Scripts that used these `mongo` shell methods for bulk write operations with "fire-and-forget" behavior should use the `Bulk()` methods.

In sharded environments, applications using any driver or `mongo` shell should use `Bulk()` methods for optimal performance when inserting or modifying groups of documents.

For example, instead of:

```
for (var i = 1; i <= 1000000; i++) {
    db.test.insert( { x : i } );
}
```

In MongoDB 2.6, replace with `Bulk()` operation:

```
var bulk = db.test.initializeUnorderedBulkOp();

for (var i = 1; i <= 1000000; i++) {
    bulk.insert( { x : i} );
}

bulk.execute( { w: 1 } );
```

Bulk method returns a `BulkWriteResult` object that contains the result of the operation.

See also:

New Write Operation Protocol (page 832), `Bulk()`, `Bulk.execute()`, `db.collection.initializeUnorderedBulkOp()`, `db.collection.initializeOrderedBulkOp()`

`db.collection.aggregate()` Change

Description The `db.collection.aggregate()` method in the `mongo` shell defaults to returning a cursor to the results set. This change enables the aggregation pipeline to return result sets of any size and requires cursor iteration to access the result set. For example:

```
var myCursor = db.orders.aggregate( [
    {
      $group: {
         _id: "$cust_id",
         total: { $sum: "$price" }
      }
    }
] );

myCursor.forEach( function(x) { printjson (x); } );
```

Previous versions returned a single document with a field `results` that contained an array of the result set, subject to the *BSON Document size* limit. Accessing the result set in the previous versions of MongoDB required accessing the `results` field and iterating the array. For example:

```
var returnedDoc = db.orders.aggregate( [
    {
      $group: {
         _id: "$cust_id",
         total: { $sum: "$price" }
      }
    }
] );

var myArray = returnedDoc.result; // access the result field
```

```
myArray.forEach( function(x) { printjson (x); } );
```

Solution Update scripts that currently expect `db.collection.aggregate()` to return a document with a `results` array to handle cursors instead.

See also:

Aggregation Enhancements (page 831), `db.collection.aggregate()`,

Write Concern Validation

Description Specifying a write concern that includes `j: true` to a `mongod` or `mongos` instance running with `--nojournal` option now errors. Previous versions would ignore the `j: true`.

Solution Either remove the `j: true` specification from the write concern when issued against a `mongod` or `mongos` instance with `--nojournal` or run `mongod` or `mongos` with journaling.

Security Changes

New Authorization Model

Description MongoDB 2.6 *authorization model* (page 320) changes how MongoDB stores and manages user privilege information:

- Before the upgrade, MongoDB 2.6 requires at least one user in the admin database.
- MongoDB versions using older models cannot create/modify users or create user-defined roles.

Solution Ensure that at least one user exists in the admin database. If no user exists in the admin database, add a user. Then upgrade to MongoDB 2.6. Finally, upgrade the user privilege model. See *Upgrade MongoDB to 2.6* (page 847).

Important: Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` (page 411) **before** upgrading the MongoDB binaries.

See also:

Security Improvements (page 833)

TLS/SSL Certificate Hostname Validation

Description The TLS/SSL certificate validation now checks the Common Name (`CN`) and the Subject Alternative Name (`SAN`) fields to ensure that either the `CN` or one of the `SAN` entries matches the hostname of the server. As a result, if you currently use TLS/SSL and *neither* the `CN` nor any of the `SAN` entries of your current TLS/SSL certificates match the hostnames, upgrading to version 2.6 will cause the TLS/SSL connections to fail.

Solution To allow for the continued use of these certificates, MongoDB provides the `allowInvalidCertificates` setting. The setting is available for:

- `mongod` and `mongos` to bypass the validation of TLS/SSL certificates on other servers in the cluster.
- `mongo` shell, *MongoDB tools that support TLS/SSL* (page 345), and the C++ driver to bypass the validation of server certificates.

When using the `allowInvalidCertificates` setting, MongoDB logs as a warning the use of the invalid certificates.

> **Warning:** The `allowInvalidCertificates` setting bypasses the other certificate validation, such as checks for expiration and valid signatures.

`2dsphere` Index Version 2

Description MongoDB 2.6 introduces a version 2 of the *2dsphere index* (page 497). If a document lacks a `2dsphere` index field (or the field is `null` or an empty array), MongoDB does not add an entry for the document to the `2dsphere` index. For inserts, MongoDB inserts the document but does not add to the `2dsphere` index.

Previous version would not insert documents where the `2dsphere` index field is a `null` or an empty array. For documents that lack the `2dsphere` index field, previous versions would insert and index the document.

Solution To revert to old behavior, create the `2dsphere` index with `{ "2dsphereIndexVersion" : 1 }` to create a version 1 index. However, version 1 index cannot use the new GeoJSON geometries.

See also:

2dsphere (Version 2) (page 497)

Log Messages

Timestamp Format Change

Description Each message now starts with the timestamp format given in *Time Format Changes* (page 846). Previous versions used the `ctime` format.

Solution MongoDB adds a new option *`--timeStampFormat`* which supports timestamp format in *`ctime`*, *`iso8601-utc`*, and *`iso8601-local`* (new default).

Package Configuration Changes

Default `bindIp` for RPM/DEB Packages

Description In the official MongoDB packages in RPM (Red Hat, CentOS, Fedora Linux, and derivatives) and DEB (Debian, Ubuntu, and derivatives), the default `bindIp` value attaches MongoDB components to the localhost interface *only*. These packages set this default in the default configuration file (i.e. `/etc/mongod.conf`.)

Solution If you use one of these packages and have *not* modified the default `/etc/mongod.conf` file, you will need to set `bindIp` before or during the upgrade.

There is no default `bindIp` setting in any other official MongoDB packages.

SNMP Changes

Description

- The IANA enterprise identifier for MongoDB changed from 37601 to 34601.
- MongoDB changed the MIB field name `globalopcounts` to `globalOpcounts`.

Solution

- Users of SNMP monitoring must modify their SNMP configuration (i.e. MIB) from 37601 to 34601.
- Update references to `globalopcounts` to `globalOpcounts`.

Remove Method Signature Change

Description `db.collection.remove()` requires a query document as a parameter. In previous versions, the method invocation without a query document deleted all documents in a collection.

Solution For existing `db.collection.remove()` invocations without a query document, modify the invocations to include an empty document `db.collection.remove({})`.

Update Operator Syntax Validation

Description

- `Update operators (e.g $set)` must specify a non-empty operand expression. For example, the following expression is now invalid:

  ```
  { $set: { } }
  ```

- `Update operators (e.g $set)` cannot repeat in the update statement. For example, the following expression is invalid:

  ```
  { $set: { a: 5 }, $set: { b: 5 } }
  ```

Updates Enforce Field Name Restrictions

Description

- Updates cannot use `update operators (e.g $set)` to target fields with empty field names (i.e. `""`).
- Updates no longer support saving field names that contain a dot (`.`) or a field name that starts with a dollar sign (`$`).

Solution

- For existing documents that have fields with empty names `""`, replace the whole document. See `db.collection.update()` and `db.collection.save()` for details on replacing an existing document.
- For existing documents that have fields with names that contain a dot (`.`), either replace the whole document or `unset` the field. To find fields whose names contain a dot, run `db.upgradeCheckAllDBs()`.
- For existing documents that have fields with names that start with a dollar sign (`$`), `unset` or `rename` those fields. To find fields whose names start with a dollar sign, run `db.upgradeCheckAllDBs()`.

See *New Write Operation Protocol* (page 832) for the changes to the write operation protocol, and *Insert and Update Improvements* (page 832) for the changes to the insert and update operations. Also consider the documentation of the `Restrictions on Field Names`.

Query and Sort Changes

Enforce Field Name Restrictions

Description Queries cannot specify conditions on fields with names that start with a dollar sign (`$`).

Solution `Unset` or `rename` existing fields whose names start with a dollar sign (`$`). Run `db.upgradeCheckAllDBs()` to find fields whose names start with a dollar sign.

Sparse Index and Incomplete Results

Description If a *sparse index* (page 507) results in an incomplete result set for queries and sort operations, MongoDB will not use that index unless a `hint()` explicitly specifies the index.

For example, the query `{ x: { $exists: false } }` will no longer use a sparse index on the `x` field, unless explicitly hinted.

Solution To override the behavior to use the sparse index and return incomplete results, explicitly specify the index with a `hint()`.

See *Sparse Index On A Collection Cannot Return Complete Results* (page 508) for an example that details the new behavior.

`sort()` Specification Values

Description The `sort()` method **only** accepts the following values for the sort keys:

- `1` to specify ascending order for a field,
- `-1` to specify descending order for a field, or
- `$meta` expression to specify sort by the text search score.

Any other value will result in an error.

Previous versions also accepted either `true` or `false` for ascending.

Solution Update sort key values that use `true` or `false` to `1`.

`skip()` and `_id` Queries

Description Equality match on the `_id` field obeys `skip()`.

Previous versions ignored `skip()` when performing an equality match on the `_id` field.

`explain()` Retains Query Plan Cache

Description `explain()` no longer clears the *query plans* (page 72) cached for that *query shape*.

In previous versions, `explain()` would have the side effect of clearing the query plan cache for that query shape.

See also:

The `PlanCache()` reference.

Geospatial Changes

`$maxDistance` Changes

Description

- For `$near` queries on GeoJSON data, if the queries specify a `$maxDistance`, `$maxDistance` must be inside of the `$near` document.

  In previous version, `$maxDistance` could be either inside or outside the `$near` document.

- `$maxDistance` must be a positive value.

Solution

- Update any existing `$near` queries on GeoJSON data that currently have the `$maxDistance` outside the `$near` document
- Update any existing queries where `$maxDistance` is a negative value.

Deprecated `$uniqueDocs`

Description MongoDB 2.6 deprecates `$uniqueDocs`, and geospatial queries no longer return duplicated results when a document matches the query multiple times.

Stronger Validation of Geospatial Queries

Description MongoDB 2.6 enforces a stronger validation of geospatial queries, such as validating the options or GeoJSON specifications, and errors if the geospatial query is invalid. Previous versions allowed/ignored invalid options.

Query Operator Changes

`$not` Query Behavior Changes

Description

- Queries with `$not` expressions on an indexed field now match:
  - Documents that are missing the indexed field. Previous versions would not return these documents using the index.
  - Documents whose indexed field value is a different type than that of the specified value. Previous versions would not return these documents using the index.

  For example, if a collection `orders` contains the following documents:

  ```
  { _id: 1, status: "A", cust_id: "123", price: 40 }
  { _id: 2, status: "A", cust_id: "xyz", price: "N/A" }
  { _id: 3, status: "D", cust_id: "xyz" }
  ```

  If the collection has an index on the `price` field:

  ```
  db.orders.ensureIndex( { price: 1 } )
  ```

  The following query uses the index to search for documents where `price` is not greater than or equal to `50`:

  ```
  db.orders.find( { price: { $not: { $gte: 50 } } } )
  ```

  In 2.6, the query returns the following documents:

  ```
  { "_id" : 3, "status" : "D", "cust_id" : "xyz" }
  { "_id" : 1, "status" : "A", "cust_id" : "123", "price" : 40 }
  { "_id" : 2, "status" : "A", "cust_id" : "xyz", "price" : "N/A" }
  ```

  In previous versions, indexed plans would only return matching documents where the type of the field matches the type of the query predicate:

  ```
  { "_id" : 1, "status" : "A", "cust_id" : "123", "price" : 40 }
  ```

  If using a collection scan, previous versions would return the same results as those in 2.6.

- MongoDB 2.6 allows chaining of `$not` expressions.

`null` Comparison Queries

Description

- `$lt` and `$gt` comparisons to `null` no longer match documents that are missing the field.
- `null` equality conditions on array elements (e.g. `"a.b": null`) no longer match document missing the nested field `a.b` (e.g. `a: [ 2, 3 ]`).
- `null` equality queries (i.e. `field: null`) now match fields with values `undefined`.

`$all` Operator Behavior Change

Description

- The `$all` operator is now equivalent to an `$and` operation of the specified values. This change in behavior can allow for more matches than previous versions when passed an array of a single nested array (e.g. `[ [ "A" ] ]`). When passed an array of a nested array, `$all` can now match documents where the field contains the nested array as an element (e.g. `field: [ [ "A" ], ... ]`), *or* the field equals the nested array (e.g. `field: [ "A", "B" ]`). Earlier version could only match documents where the field contains the nested array.
- The `$all` operator returns no match if the array field contains nested arrays (e.g. `field: [ "a", ["b"] ]`) and `$all` on the nested field is the element of the nested array (e.g. `"field.1": { $all: [ "b" ] }`). Previous versions would return a match.

`$mod` Operator Enforces Strict Syntax

Description The `$mod` operator now only accepts an array with exactly two elements, and errors when passed an array with fewer or more elements. See *mod-not-enough-elements* and *mod-too-many-elements* for details.

In previous versions, if passed an array with one element, the `$mod` operator uses `0` as the second element, and if passed an array with more than two elements, the `$mod` ignores all but the first two elements. Previous versions do return an error when passed an empty array.

Solution Ensure that the array passed to `$mod` contains exactly two elements:

- If the array contains the a single element, add `0` as the second element.
- If the array contains more than two elements, remove the extra elements.

`$where` Must Be Top-Level

Description `$where` expressions can now only be at top level and cannot be nested within another expression, such as `$elemMatch`.

Solution Update existing queries that nest `$where`.

`$exists` and `notablescan` If the MongoDB server has disabled collection scans, i.e. `notablescan`, then `$exists` queries that have no *indexed solution* will error.

`MinKey` and `MaxKey` Queries

Description Equality match for either `MinKey` or `MaxKey` no longer match documents missing the field.

Nested Array Queries with $elemMatch

Description The `$elemMatch` query operator no longer traverses recursively into nested arrays.

For example, if a collection `test` contains the following document:

```
{ "_id": 1, "a" : [ [ 1, 2, 5 ] ] }
```

In 2.6, the following `$elemMatch` query does *not* match the document:

```
db.test.find( { a: { $elemMatch: { $gt: 1, $lt: 5 } } } )
```

Solution Update existing queries that rely upon the old behavior.

Text Search Compatibility MongoDB does not support the use of the `$text` query operator in mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards. See *Upgrade MongoDB to 2.6* (page 847) for upgrade instructions.

Replica Set/Sharded Cluster Validation

Shard Name Checks on Metadata Refresh

Description For sharded clusters, MongoDB 2.6 disallows a shard from refreshing the metadata if the shard name has not been explicitly set.

For mixed sharded cluster deployments that contain both version 2.4 and version 2.6 shards, this change can cause errors when migrating chunks **from** version 2.4 shards **to** version 2.6 shards if the shard name is unknown to the version 2.6 shards. MongoDB does not support migrations in mixed sharded cluster deployments.

Solution Upgrade all components of the cluster to 2.6. See *Upgrade MongoDB to 2.6* (page 847).

Replica Set Vote Configuration Validation

Description MongoDB now deprecates giving any *replica set* member more than a single vote. During configuration, `local.system.replset.members[n].votes` (page 663) should only have a value of 1 for voting members and 0 for non-voting members. MongoDB treats values other than 1 or 0 as a value of 1 and produces a warning message.

Solution Update `local.system.replset.members[n].votes` (page 663) with values other than 1 or 0 to 1 or 0 as appropriate.

Time Format Changes MongoDB now uses `iso8601-local` when formatting time data in many outputs. This format follows the template `YYYY-MM-DDTHH:mm:ss.mmm<+/-Offset>`. For example, `2014-03-04T20:13:38.944-0500`.

This change impacts all clients using `Extended JSON` in *Strict mode*, such as `mongoexport` and the REST and HTTP Interfaces[404].

Other Resources

- All backwards incompatible changes (JIRA)[405].
- *Release Notes for MongoDB 2.6* (page 805).

[404] https://docs.mongodb.org/ecosystem/tools/http-interfaces

[405] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.5.0%22%2C%20%222.5.1%22%2C%20%222.5.2%22
rc1%22%2C%20%222.6.0-rc2%22)%20AND%20%22Backwards%20Compatibility%22%20in%20%20(%22Major%20Change%22%2C%20%22Minor%20Change%2

- *Upgrade MongoDB to 2.6* (page 847) for the upgrade process.

Some changes in 2.6 can affect *compatibility* (page 836) and may require user actions. The 2.6 `mongo` shell provides a `db.upgradeCheckAllDBs()` method to perform a check for upgrade preparedness for some of these changes.

See *Compatibility Changes in MongoDB 2.6* (page 836) for a detailed list of compatibility changes.

See also:

All Backwards incompatible changes (JIRA)[406].

Upgrade Process

	On this page
Upgrade MongoDB to 2.6	• Upgrade Recommendations and Checklists (page 847) • Upgrade MongoDB Processes (page 848) • Upgrade Procedure (page 850) • Additional Resources (page 851)

In the general case, the upgrade from MongoDB 2.4 to 2.6 is a binary-compatible "drop-in" upgrade: shut down the `mongod` instances and replace them with `mongod` instances running 2.6. **However**, before you attempt any upgrade, familiarize yourself with the content of this document, particularly the *Upgrade Recommendations and Checklists* (page 847), the procedure for *upgrading sharded clusters* (page 849), and the considerations for *reverting to 2.4 after running 2.6* (page 853).

Upgrade Recommendations and Checklists When upgrading, consider the following:

Upgrade Requirements To upgrade an existing MongoDB deployment to 2.6, you must be running 2.4. If you're running a version of MongoDB before 2.4, you *must* upgrade to 2.4 before upgrading to 2.6. See *Upgrade MongoDB to 2.4* (page 874) for the procedure to upgrade from 2.2 to 2.4.

If you use MongoDB Cloud Manager[407] Backup, ensure that you're running *at least* version `v20131216.1` of the Backup agent before upgrading. Version `1.4.0` of the backup agent followed `v20131216.1`

Preparedness Before upgrading MongoDB always test your application in a staging environment before deploying the upgrade to your production environment.

To begin the upgrade procedure, connect a 2.6 `mongo` shell to your MongoDB 2.4 `mongos` or `mongod` and run the `db.upgradeCheckAllDBs()` to check your data set for compatibility. This is a preliminary automated check. Assess and resolve all issues identified by `db.upgradeCheckAllDBs()`.

Some changes in MongoDB 2.6 require manual checks and intervention. See *Compatibility Changes in MongoDB 2.6* (page 836) for an explanation of these changes. Resolve all incompatibilities in your deployment before continuing.

For a deployment that uses authentication and authorization, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` (page 411) **before** upgrading the MongoDB binaries. For deployments currently using authentication and authorization, see the *consideration for deployments that use authentication and authorization* (page 848).

[406] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.5.0%22%2C%20%222.5.1%22%2C%20%222.5.2%22
rc1%22%2C%20%222.6.0-rc2%22%2C%20%222.6.0-rc3%22)%20AND%20%22Backwards%20Compatibility%22%20in%20(%20%22Minor%20Change%22%2C%2
[407] https://cloud.mongodb.com/?jmp=docs

Authentication MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

Before beginning the upgrade process for a deployment that uses authentication and authorization:

- Ensure that at least one user exists in the `admin` database with the role `userAdminAnyDatabase` (page 411).
- If your application performs CRUD operations on the `<database>.system.users` collection or uses a `db.addUser()`-like method, then you **must** upgrade those drivers (i.e. client libraries) **before** `mongod` or `mongos` instances.
- You must fully complete the upgrade procedure for *all* MongoDB processes before upgrading the authorization model.

After you begin to upgrade a MongoDB deployment that uses authentication to 2.6, you *cannot* modify existing user data until you complete the *authorization user schema upgrade* (page 851).

See *Upgrade User Authorization Data to 2.6 Format* (page 851) for a complete discussion of the upgrade procedure for the authorization model including additional requirements and procedures.

Downgrade Limitations Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you created `text` or `2dsphere` indexes while running 2.6, you can only downgrade to MongoDB 2.4.10 or later.

Package Upgrades If you installed MongoDB from the MongoDB `apt` or `yum` repositories, upgrade to 2.6 using the package manager.

For Debian, Ubuntu, and related operating systems, type these commands:

```
sudo apt-get update
sudo apt-get install mongodb-org
```

For Red Hat Enterprise, CentOS, Fedora, or Amazon Linux:

```
sudo yum install mongodb-org
```

If you did not install the `mongodb-org` package, and installed a subset of MongoDB components replace `mongodb-org` in the commands above with the appropriate package names.

See installation instructions for *Ubuntu* (page 10), *RHEL* (page 6), *Debian* (page 13), or *other Linux Systems* (page 16) for a list of the available packages and complete MongoDB installation instructions.

Upgrade MongoDB Processes

Upgrade Standalone `mongod` Instance to MongoDB 2.6 The following steps outline the procedure to upgrade a standalone `mongod` from version 2.4 to 2.6. To upgrade from version 2.2 to 2.6, *upgrade to version 2.4* (page 874) *first*, and then follow the procedure to upgrade from 2.4 to 2.6.

1. Download binaries of the latest release in the 2.6 series from the MongoDB Download Page[408]. See *Install MongoDB* (page 5) for more information.
2. Shut down your `mongod` instance. Replace the existing binary with the 2.6 `mongod` binary and restart `mongod`.

[408] http://www.mongodb.org/downloads

Upgrade a Replica Set to 2.6 The following steps outline the procedure to upgrade a replica set from MongoDB 2.4 to MongoDB 2.6. To upgrade from MongoDB 2.2 to 2.6, *upgrade all members of the replica set to version 2.4* (page 874) *first*, and then follow the procedure to upgrade from MongoDB 2.4 to 2.6.

You can upgrade from MongoDB 2.4 to 2.6 using a "rolling" upgrade to minimize downtime by upgrading the members individually while the other members are available:

Step 1: Upgrade secondary members of the replica set. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 2.4 binary with the 2.6 binary. After upgrading a `mongod` instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.

Step 2: Step down the replica set primary. Use `rs.stepDown()` in the `mongo` shell to step down the *primary* and force the set to *failover* (page 583). `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Upgrade the primary. When `rs.status()` shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` binary with the 2.6 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable accept writes until the failover process completes. Typically this takes 30 seconds or more: schedule the upgrade procedure during a scheduled maintenance window.

Upgrade a Sharded Cluster to 2.6 Only upgrade sharded clusters to 2.6 if **all** members of the cluster are currently running instances of 2.4. The only supported upgrade path for sharded clusters running 2.2 is via 2.4. The upgrade process checks all components of the cluster and will produce warnings if any component is running version 2.2.

Considerations The upgrade process does not require any downtime. However, while you upgrade the sharded cluster, ensure that clients do not make changes to the collection meta-data. For example, during the upgrade, do **not** do any of the following:

- `sh.enableSharding()`
- `sh.shardCollection()`
- `sh.addShard()`
- `db.createCollection()`
- `db.collection.drop()`
- `db.dropDatabase()`
- any operation that creates a database
- any other operation that modifies the cluster metadata in any way. See *Sharding Reference* (page 753) for a complete list of sharding commands. Note, however, that not all commands on the *Sharding Reference* (page 753) page modifies the cluster meta-data.

Upgrade Sharded Clusters *Optional but Recommended.* As a precaution, take a backup of the `config` database *before* upgrading the sharded cluster.

Step 1: Disable the Balancer. Turn off the *balancer* (page 698) in the sharded cluster, as described in *Disable the Balancer* (page 732).

Step 2: Upgrade the cluster's meta data. Start a single 2.6 `mongos` instance with the `configDB` pointing to the cluster's config servers and with the `--upgrade` option.

To run a `mongos` with the `--upgrade` option, you can upgrade an existing `mongos` instance to 2.6, or if you need to avoid reconfiguring a production `mongos` instance, you can use a new 2.6 `mongos` that can reach all the config servers.

To upgrade the meta data, run:

```
mongos --configdb <configDB string> --upgrade
```

You can include the `--logpath` option to output the log messages to a file instead of the standard output. Also include any other options required to start `mongos` instances in your cluster, such as `--sslOnNormalPorts` or `--sslPEMKeyFile`.

The `mongos` will exit upon completion of the `--upgrade` process.

The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If the data files have many sharded collections or if failed processes hold stale locks, acquiring the locks for all collections can take seconds or minutes. Watch the log for progress updates.

Step 3: Ensure `mongos --upgrade` process completes successfully. The `mongos` will exit upon completion of the meta data upgrade process. If successful, the process will log the following messages:

```
upgrade of config server to v5 successful
Config database is at version v5
```

After a successful upgrade, restart the `mongos` instance. If `mongos` fails to start, check the log for more information.

If the `mongos` instance loses its connection to the config servers during the upgrade or if the upgrade is otherwise unsuccessful, you may always safely retry the upgrade.

Step 4: Upgrade the remaining `mongos` instances to v2.6. Upgrade and restart **without** the `--upgrade` option the other `mongos` instances in the sharded cluster. After upgrading all the `mongos`, see *Complete Sharded Cluster Upgrade* (page 850) for information on upgrading the other cluster components.

Complete Sharded Cluster Upgrade After you have successfully upgraded *all* `mongos` instances, you can upgrade the other instances in your MongoDB deployment.

> **Warning:** Do not upgrade `mongod` instances until after you have upgraded *all* `mongos` instances.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all 3 `mongod` config server instances, leaving the *first* system in the `mongos --configdb` argument to upgrade *last*.
- Upgrade each shard, one at a time, upgrading the `mongod` secondaries before running `replSetStepDown` and upgrading the primary of each shard.

When this process is complete, *re-enable the balancer* (page 733).

Upgrade Procedure Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you have `text` or `2dsphere` indexes, you can only downgrade to MongoDB 2.4.10 or later.

Except as described on this page, moving between 2.4 and 2.6 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.4 `mongod` with the *`--shutdown`* option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *Stop mongod Processes* (page 237) for alternate methods of stopping a `mongod` instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.6 `mongod` with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

Additional Resources

- MongoDB Major Version Upgrade Consulting Package[409]

Upgrade User Authorization Data to 2.6 Format

On this page

MongoDB 2.6 includes significant changes to the authorization model, which requires changes to the way that MongoDB stores users' credentials. As a result, in addition to upgrading MongoDB processes, if your deployment uses authentication and authorization, after upgrading all MongoDB process to 2.6 you **must** also upgrade the authorization model.

Considerations

Complete all other Upgrade Requirements Before upgrading the authorization model, you should first upgrade MongoDB binaries to 2.6. For sharded clusters, ensure that **all** cluster components are 2.6. If there are users in any database, be sure you have at least one user in the `admin` database with the role `userAdminAnyDatabase` (page 411) **before** upgrading the MongoDB binaries.

Timing Because downgrades are more difficult after you upgrade the user authorization model, once you upgrade the MongoDB binaries to version 2.6, allow your MongoDB deployment to run a day or two **without** upgrading the user authorization model.

This allows 2.6 some time to "burn in" and decreases the likelihood of downgrades occurring after the user privilege model upgrade. The user authentication and access control will continue to work as it did in 2.4, **but** it will be impossible to create or modify users or to use user-defined roles until you run the authorization upgrade.

If you decide to upgrade the user authorization model immediately instead of waiting the recommended "burn in" period, then for sharded clusters, you must wait at least 10 seconds after upgrading the sharded clusters to run the authorization upgrade script.

[409] https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade

Replica Sets For a replica set, it is only necessary to run the upgrade process on the *primary* as the changes will automatically replicate to the secondaries.

Sharded Clusters For a sharded cluster, connect to a `mongos` and run the upgrade procedure to upgrade the cluster's authorization data. By default, the procedure will upgrade the authorization data of the shards as well.

To override this behavior, run the upgrade command with the additional parameter `upgradeShards: false`. If you choose to override, you must run the upgrade procedure on the `mongos` first, and then run the procedure on the *primary* members of each shard.

For a sharded cluster, do **not** run the upgrade process directly against the *config servers* (page 684). Instead, perform the upgrade process using one `mongos` instance to interact with the config database.

Requirements To upgrade the authorization model, you must have a user in the `admin` database with the role `userAdminAnyDatabase` (page 411).

Procedure

Step 1: Connect to MongoDB instance. Connect and authenticate to the `mongod` instance for a single deployment or a `mongos` for a sharded cluster as an `admin` database user with the role `userAdminAnyDatabase` (page 411).

Step 2: Upgrade authorization schema. Use the `authSchemaUpgrade` command in the `admin` database to update the user data using the `mongo` shell.

Run `authSchemaUpgrade` command.

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1 });
```

In case of error, you may safely rerun the `authSchemaUpgrade` command.

Sharded cluster `authSchemaUpgrade` consideration. For a sharded cluster, `authSchemaUpgrade` will upgrade the authorization data of the shards as well and the upgrade is complete. You can, however, override this behavior by including `upgradeShards: false` in the command, as in the following example:

```
db.getSiblingDB("admin").runCommand({authSchemaUpgrade: 1,
upgradeShards: false });
```

If you override the behavior, after running `authSchemaUpgrade` on a `mongos` instance, you will need to connect to the primary for each shard and repeat the upgrade process after upgrading on the `mongos`.

Result All users in a 2.6 system are stored in the `admin.system.users` (page 304) collection. To manipulate these users, use the `user management methods`.

The upgrade procedure copies the version 2.4 `admin.system.users` collection to `admin.system.backup_users`.

The upgrade procedure leaves the version 2.4 `<database>.system.users` collection(s) intact.

Downgrade MongoDB from 2.6

On this page

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the *Downgrade Recommendations and Checklist* (page 853) and the procedure for *downgrading sharded clusters* (page 857).

Downgrade Recommendations and Checklist When downgrading, consider the following:

Downgrade Path Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you created `text` or `2dsphere` indexes while running 2.6, you can only downgrade to MongoDB 2.4.10 or later.

Preparedness

- *Remove or downgrade version 2 text indexes* (page 856) before downgrading MongoDB 2.6 to 2.4.
- *Remove or downgrade version 2 2dsphere indexes* (page 856) before downgrading MongoDB 2.6 to 2.4.
- *Downgrade 2.6 User Authorization Model* (page 853). If you have upgraded to the 2.6 user authorization model, you must downgrade the user model to 2.4 before downgrading MongoDB 2.6 to 2.4.

Procedures Follow the downgrade procedures:

- To downgrade sharded clusters, see *Downgrade a 2.6 Sharded Cluster* (page 857).
- To downgrade replica sets, see *Downgrade a 2.6 Replica Set* (page 857).
- To downgrade a standalone MongoDB instance, see *Downgrade 2.6 Standalone mongod Instance* (page 857).

Downgrade 2.6 User Authorization Model If you have upgraded to the 2.6 user authorization model, you **must first** downgrade the user authorization model to 2.4 **before** before downgrading MongoDB 2.6 to 2.4.

Considerations

- For a replica set, it is only necessary to run the downgrade process on the *primary* as the changes will automatically replicate to the secondaries.
- For sharded clusters, although the procedure lists the downgrade of the cluster's authorization data first, you may downgrade the authorization data of the cluster or shards first.
- You *must* have the `admin.system.backup_users` and `admin.system.new_users` collections created during the upgrade process.
- **Important**. The downgrade process returns the user data to its state prior to upgrading to 2.6 authorization model. Any changes made to the user/role data using the 2.6 users model will be lost.

Access Control Prerequisites To downgrade the authorization model, you must connect as a user with the following *privileges*:

```
{ resource: { db: "admin", collection: "system.new_users" }, actions: [ "find", "insert", "update" ]
{ resource: { db: "admin", collection: "system.backup_users" }, actions: [  "find" ] }
{ resource: { db: "admin", collection: "system.users" }, actions: [ "find", "remove", "insert"] }
{ resource: { db: "admin", collection: "system.version" }, actions: [ "find", "update" ] }
```

If no user exists with the appropriate *privileges*, create an authorization model downgrade user:

Step 1: Connect as user with privileges to manage users and roles. Connect and authenticate as a user with `userAdminAnyDatabase` (page 411).

Step 2: Create a role with required privileges. Using the `db.createRole` method, create a *role* (page 321) with the required privileges.

```
use admin
db.createRole(
  {
    role: "downgradeAuthRole",
    privileges: [
      { resource: { db: "admin", collection: "system.new_users" }, actions: [ "find", "insert", "upda
      { resource: { db: "admin", collection: "system.backup_users" }, actions: [  "find" ] },
      { resource: { db: "admin", collection: "system.users" }, actions: [ "find", "remove", "insert"
      { resource: { db: "admin", collection: "system.version" }, actions: [ "find", "update" ] }
    ],
    roles: [ ]
  }
)
```

Step 3: Create a user with the new role. Create a user and assign the user the `downgradeRole`.

```
use admin
db.createUser(
   {
     user: "downgradeAuthUser",
     pwd: "somePass123",
     roles: [ { role: "downgradeAuthRole", db: "admin" } ]
   }
)
```

Note: Instead of creating a new user, you can also grant the role to an existing user. See `db.grantRolesToUser()` method.

Step 4: Authenticate as the new user. Authenticate as the newly created user.

```
use admin
db.auth( "downgradeAuthUser", "somePass123" )
```

The method returns `1` upon successful authentication.

Procedure The following downgrade procedure requires `<database>.system.users` collections used in version 2.4. to be intact for non-`admin` databases.

Step 1: Connect and authenticate to MongoDB instance. Connect and authenticate to the `mongod` instance for a single deployment or a `mongos` for a sharded cluster with the appropriate privileges. See *Access Control Prerequisites* (page 854) for details.

Step 2: Create backup of 2.6 `admin.system.users` collection. Copy all documents in the `admin.system.users` (page 304) collection to the `admin.system.new_users` collection:

```
db.getSiblingDB("admin").system.users.find().forEach( function(userDoc) {
    status = db.getSiblingDB("admin").system.new_users.save( userDoc );
    if (status.hasWriteError()) {
        print(status.writeError);
    }
  }
);
```

Step 3: Update the version document for the `authSchema`.

```
db.getSiblingDB("admin").system.version.update(
   { _id: "authSchema" },
   { $set: { currentVersion: 2 } }
);
```

The method returns a `WriteResult` object with the status of the operation. Upon successful update, the `WriteResult` object should have `"nModified"` equal to `1`.

Step 4: Remove existing documents from the `admin.system.users` collection.

```
db.getSiblingDB("admin").system.users.remove( {} )
```

The method returns a `WriteResult` object with the number of documents removed in the `"nRemoved"` field.

Step 5: Copy documents from the `admin.system.backup_users` collection. Copy all documents from the `admin.system.backup_users`, created during the 2.6 upgrade, to `admin.system.users`.

```
db.getSiblingDB("admin").system.backup_users.find().forEach(
   function (userDoc) {
      status = db.getSiblingDB("admin").system.users.insert( userDoc );
      if (status.hasWriteError()) {
          print(status.writeError);
      }
   }
);
```

Step 6: Update the version document for the `authSchema`.

```
db.getSiblingDB("admin").system.version.update(
   { _id: "authSchema" },
   { $set: { currentVersion: 1 } }
)
```

For a sharded cluster, repeat the downgrade process by connecting to the *primary* replica set member for each shard.

Note: The cluster's `mongos` instances will fail to detect the authorization model downgrade until the user cache is refreshed. You can run `invalidateUserCache` on each `mongos` instance to refresh immediately, or you can wait until the cache is refreshed automatically at the end of the `user cache invalidation interval`. To

run `invalidateUserCache`, you must have privilege with `invalidateUserCache` (page 420) action, which is granted by `userAdminAnyDatabase` (page 411) and `hostManager` (page 409) roles.

Result The downgrade process returns the user data to its state prior to upgrading to 2.6 authorization model. Any changes made to the user/role data using the 2.6 users model will be lost.

Downgrade Updated Indexes

Text Index Version Check If you have *version 2* text indexes (i.e. the default version for text indexes in MongoDB 2.6), drop the *version 2* text indexes before downgrading MongoDB. After the downgrade, enable text search and recreate the dropped text indexes.

To determine the version of your `text` indexes, run `db.collection.getIndexes()` to view index specifications. For text indexes, the method returns the version information in the field `textIndexVersion`. For example, the following shows that the `text` index on the `quotes` collection is version 2.

```
{
   "v" : 1,
   "key" : {
      "_fts" : "text",
      "_ftsx" : 1
   },
   "name" : "quote_text_translation.quote_text",
   "ns" : "test.quotes",
   "weights" : {
      "quote" : 1,
      "translation.quote" : 1
   },
   "default_language" : "english",
   "language_override" : "language",
   "textIndexVersion" : 2
}
```

`2dsphere` Index Version Check If you have *version 2* `2dsphere` indexes (i.e. the default version for `2dsphere` indexes in MongoDB 2.6), drop the *version 2* `2dsphere` indexes before downgrading MongoDB. After the downgrade, recreate the `2dsphere` indexes.

To determine the version of your `2dsphere` indexes, run `db.collection.getIndexes()` to view index specifications. For `2dsphere` indexes, the method returns the version information in the field `2dsphereIndexVersion`. For example, the following shows that the `2dsphere` index on the `locations` collection is version 2.

```
{
   "v" : 1,
   "key" : {
      "geo" : "2dsphere"
   },
   "name" : "geo_2dsphere",
   "ns" : "test.locations",
   "sparse" : true,
   "2dsphereIndexVersion" : 2
}
```

Downgrade MongoDB Processes

Downgrade 2.6 Standalone `mongod` Instance The following steps outline the procedure to downgrade a standalone `mongod` from version 2.6 to 2.4.

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page[410]. See *Install MongoDB* (page 5) for more information.
2. Shut down your `mongod` instance. Replace the existing binary with the 2.4 `mongod` binary and restart `mongod`.

Downgrade a 2.6 Replica Set The following steps outline a "rolling" downgrade process for the replica set. The "rolling" downgrade process minimizes downtime by downgrading the members individually while the other members are available:

Step 1: Downgrade each secondary member, one at a time. For each *secondary* in a replica set:

Replace and restart secondary `mongod` instances. First, shut down the `mongod`, then replace these binaries with the 2.4 binary and restart `mongod`. See *Stop mongod Processes* (page 237) for instructions on safely terminating `mongod` processes.

Allow secondary to recover. Wait for the member to recover to `SECONDARY` state before upgrading the next secondary.

To check the member's state, use the `rs.status()` method in the `mongo` shell.

Step 2: Step down the primary. Use `rs.stepDown()` in the `mongo` shell to step down the *primary* and force the normal *failover* (page 583) procedure.

```
rs.stepDown()
```

`rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

Step 3: Replace and restart former primary `mongod`. When `rs.status()` shows that the primary has stepped down and another member has assumed `PRIMARY` state, shut down the previous primary and replace the `mongod` binary with the 2.4 binary and start the new instance.

Replica set failover is not instant but will render the set unavailable to writes and interrupt reads until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the downgrade during a predetermined maintenance window.

Downgrade a 2.6 Sharded Cluster

Requirements While the downgrade is in progress, you cannot make changes to the collection meta-data. For example, during the downgrade, do **not** do any of the following:

- `sh.enableSharding()`
- `sh.shardCollection()`
- `sh.addShard()`

[410] http://www.mongodb.org/downloads

- `db.createCollection()`
- `db.collection.drop()`
- `db.dropDatabase()`
- any operation that creates a database
- any other operation that modifies the cluster meta-data in any way. See *Sharding Reference* (page 753) for a complete list of sharding commands. Note, however, that not all commands on the *Sharding Reference* (page 753) page modifies the cluster meta-data.

Procedure The downgrade procedure for a sharded cluster reverses the order of the upgrade procedure.

1. Turn off the *balancer* (page 698) in the sharded cluster, as described in *Disable the Balancer* (page 732).
2. Downgrade each shard, one at a time. For each shard,
   (a) Downgrade the `mongod` secondaries *before* downgrading the primary.
   (b) To downgrade the primary, run `replSetStepDown` and downgrade.
3. Downgrade all 3 `mongod` config server instances, leaving the *first* system in the `mongos --configdb` argument to downgrade *last*.
4. Downgrade and restart each `mongos`, one at a time. The downgrade process is a binary drop-in replacement.
5. Turn on the balancer, as described in *Enable the Balancer* (page 733).

Downgrade Procedure Once upgraded to MongoDB 2.6, you **cannot** downgrade to **any** version earlier than MongoDB 2.4. If you have `text` or `2dsphere` indexes, you can only downgrade to MongoDB 2.4.10 or later.

Except as described on this page, moving between 2.4 and 2.6 is a drop-in replacement:

Step 1: Stop the existing `mongod` instance. For example, on Linux, run 2.6 `mongod` with the `--shutdown` option as follows:

```
mongod --dbpath /var/mongod/data --shutdown
```

Replace `/var/mongod/data` with your MongoDB `dbPath`. See also the *Stop mongod Processes* (page 237) for alternate methods of stopping a `mongod` instance.

Step 2: Start the new `mongod` instance. Ensure you start the 2.4 `mongod` with the same `dbPath`:

```
mongod --dbpath /var/mongod/data
```

Replace `/var/mongod/data` with your MongoDB `dbPath`.

See *Upgrade MongoDB to 2.6* (page 847) for full upgrade instructions.

Download

To download MongoDB 2.6, go to the downloads page[411].

[411] http://www.mongodb.org/downloads

Other Resources

- All JIRA issues resolved in 2.6[412].
- All Third Party License Notices[413].

12.2 Previous Stable Releases

12.2.1 Release Notes for MongoDB 2.4

March 19, 2013

On this page

MongoDB 2.4 includes enhanced geospatial support, switch to V8 JavaScript engine, security enhancements, and text search (beta) and hashed index.

Minor Releases

2.4 Changelog

On this page

2.4.14

- Packaging: Init script sets process ulimit to different value compared to documentation (SERVER-17780[414])
- Security: Compute BinData length in v8 (SERVER-17647[415])
- Build: Upgrade PCRE Version from 8.30 to Latest (SERVER-17252[416])

[412] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.5.0%22%2C+%222.5.1%22%2rc1%22%2C+%222.6.0-rc2%22%2C+%222.6.0-rc3%22%29

[413] https://github.com/mongodb/mongo/blob/v2.6/distsrc/THIRD-PARTY-NOTICES

[414] https://jira.mongodb.org/browse/SERVER-17780

[415] https://jira.mongodb.org/browse/SERVER-17647

[416] https://jira.mongodb.org/browse/SERVER-17252

2.4.13 - Changes

- Security: Enforce BSON BinData length validation (SERVER-17278[417])
- Security: Disable SSLv3 ciphers (SERVER-15673[418])
- Networking: Improve BSON validation (SERVER-17264[419])

2.4.12 - Changes

- Sharding: Sharded connection cleanup on setup error can crash mongos (SERVER-15056[420])
- Sharding: "type 7" (OID) error when acquiring distributed lock for first time (SERVER-13616[421])
- Storage: explicitly zero .ns files on creation (SERVER-15369[422])
- Storage: partially written journal last section causes recovery to fail (SERVER-15111[423])

2.4.11 - Changes

- Security: Potential information leak (SERVER-14268[424])
- Replication: `_id` with `$prefix` field causes replication failure due to unvalidated insert (SERVER-12209[425])
- Sharding: Invalid access: seg fault in `SplitChunkCommand::run` (SERVER-14342[426])
- Indexing: Creating descending index on `_id` can corrupt namespace (SERVER-14833[427])
- Text Search: Updates to documents with text-indexed fields may lead to incorrect entries (SERVER-14738[428])
- Build: Add SCons flag to override treating all warnings as errors (SERVER-13724[429])
- Packaging: Fix mongodb enterprise 2.4 init script to allow multiple processes per host (SERVER-14336[430])
- JavaScript: Do not store native function pointer as a property in function prototype (SERVER-14254[431])

2.4.10 - Changes

- Indexes: Fixed issue that can cause index corruption when building indexes concurrently (SERVER-12990[432])
- Indexes: Fixed issue that can cause index corruption when shutting down secondary node during index build (SERVER-12956[433])
- Indexes: Mongod now recognizes incompatible "future" text and geo index versions and exits gracefully (SERVER-12914[434])

[417] https://jira.mongodb.org/browse/SERVER-17278
[418] https://jira.mongodb.org/browse/SERVER-15673
[419] https://jira.mongodb.org/browse/SERVER-17264
[420] https://jira.mongodb.org/browse/SERVER-15056
[421] https://jira.mongodb.org/browse/SERVER-13616
[422] https://jira.mongodb.org/browse/SERVER-15369
[423] https://jira.mongodb.org/browse/SERVER-15111
[424] https://jira.mongodb.org/browse/SERVER-14268
[425] https://jira.mongodb.org/browse/SERVER-12209
[426] https://jira.mongodb.org/browse/SERVER-14342
[427] https://jira.mongodb.org/browse/SERVER-14833
[428] https://jira.mongodb.org/browse/SERVER-14738
[429] https://jira.mongodb.org/browse/SERVER-13724
[430] https://jira.mongodb.org/browse/SERVER-14336
[431] https://jira.mongodb.org/browse/SERVER-14254
[432] https://jira.mongodb.org/browse/SERVER-12990
[433] https://jira.mongodb.org/browse/SERVER-12956
[434] https://jira.mongodb.org/browse/SERVER-12914

- Indexes: Fixed issue that can cause secondaries to fail replication when building the same index multiple times concurrently (SERVER-12662[435])
- Indexes: Fixed issue that can cause index corruption on the tenth index in a collection if the index build fails (SERVER-12481[436])
- Indexes: Introduced versioning for text and geo indexes to ensure backwards compatibility (SERVER-12175[437])
- Indexes: Disallowed building indexes on the system.indexes collection, which can lead to initial sync failure on secondaries (SERVER-10231[438])
- Sharding: Avoid frequent immediate balancer retries when config servers are out of sync (SERVER-12908[439])
- Sharding: Add indexes to locks collection on config servers to avoid long queries in case of large numbers of collections (SERVER-12548[440])
- Sharding: Fixed issue that can corrupt the config metadata cache when sharding collections concurrently (SERVER-12515[441])
- Sharding: Don't move chunks created on collections with a hashed shard key if the collection already contains data (SERVER-9259[442])
- Replication: Fixed issue where node appears to be down in a replica set during a compact operation (SERVER-12264[443])
- Replication: Fixed issue that could cause delays in elections when a node is not vetoing an election (SERVER-12170[444])
- Replication: Step down all primaries if multiple primaries are detected in replica set to ensure correct election result (SERVER-10793[445])
- Replication: Upon clock skew detection, secondaries will switch to sync directly from the primary to avoid sync cycles (SERVER-8375[446])
- Runtime: The SIGXCPU signal is now caught and mongod writes a log message and exits gracefully (SERVER-12034[447])
- Runtime: Fixed issue where mongod fails to start on Linux when /sys/dev/block directory is not readable (SERVER-9248[448])
- Windows: No longer zero-fill newly allocated files on systems other than Windows 7 or Windows Server 2008 R2 (SERVER-8480[449])
- GridFS: Chunk size is decreased to 255 KB (from 256 KB) to avoid overhead with usePowerOf2Sizes option (SERVER-13331[450])
- SNMP: Fixed MIB file validation under smilint (SERVER-12487[451])

[435] https://jira.mongodb.org/browse/SERVER-12662
[436] https://jira.mongodb.org/browse/SERVER-12481
[437] https://jira.mongodb.org/browse/SERVER-12175
[438] https://jira.mongodb.org/browse/SERVER-10231
[439] https://jira.mongodb.org/browse/SERVER-12908
[440] https://jira.mongodb.org/browse/SERVER-12548
[441] https://jira.mongodb.org/browse/SERVER-12515
[442] https://jira.mongodb.org/browse/SERVER-9259
[443] https://jira.mongodb.org/browse/SERVER-12264
[444] https://jira.mongodb.org/browse/SERVER-12170
[445] https://jira.mongodb.org/browse/SERVER-10793
[446] https://jira.mongodb.org/browse/SERVER-8375
[447] https://jira.mongodb.org/browse/SERVER-12034
[448] https://jira.mongodb.org/browse/SERVER-9248
[449] https://jira.mongodb.org/browse/SERVER-8480
[450] https://jira.mongodb.org/browse/SERVER-13331
[451] https://jira.mongodb.org/browse/SERVER-12487

- Shell: Fixed issue in V8 memory allocation that could cause long-running shell commands to crash (SERVER-11871[452])
- Shell: Fixed memory leak in the md5sumFile shell utility method (SERVER-11560[453])

Previous Releases

- All 2.4.9 improvements[454].
- All 2.4.8 improvements[455].
- All 2.4.7 improvements[456].
- All 2.4.6 improvements[457].
- All 2.4.5 improvements[458].
- All 2.4.4 improvements[459].
- All 2.4.3 improvements[460].
- All 2.4.2 improvements[461]
- All 2.4.1 improvements[462].

2.4.14 – April 28, 2015

- Init script sets process ulimit to different value compared to documentation SERVER-17780[463]
- Compute BinData length in v8 SERVER-17647[464]
- Upgrade PCRE Version from 8.30 to Latest SERVER-17252[465]
- *2.4.14 Changelog* (page 859).
- All 2.4.14 improvements[466].

2.4.13 – February 25, 2015

- Enforce BSON BinData length validation SERVER-17278[467]
- Disable SSLv3 ciphers SERVER-15673[468]
- Improve BSON validation SERVER-17264[469]

[452] https://jira.mongodb.org/browse/SERVER-11871
[453] https://jira.mongodb.org/browse/SERVER-11560
[454] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
[455] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
[456] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
[457] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
[458] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
[459] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
[460] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
[461] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
[462] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
[463] https://jira.mongodb.org/browse/SERVER-17780
[464] https://jira.mongodb.org/browse/SERVER-17647
[465] https://jira.mongodb.org/browse/SERVER-17252
[466] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.14%22%20AND%20project%20%3D%20SERVER
[467] https://jira.mongodb.org/browse/SERVER-17278
[468] https://jira.mongodb.org/browse/SERVER-15673
[469] https://jira.mongodb.org/browse/SERVER-17264

- *2.4.13 Changelog* (page 860).
- All 2.4.13 improvements[470].

2.4.12 – October 16, 2014

- Partially written journal last section causes recovery to fail SERVER-15111[471].
- Explicitly zero `.ns` files on creation SERVER-15369[472].
- *2.4.12 Changelog* (page 860).
- All 2.4.12 improvements[473].

2.4.11 – August 18, 2014

- Fixed potential information leak: SERVER-14268[474].
- Resolved issue were an `_id` with a `$prefix` field caused replication failure due to unvalidated insert SERVER-12209[475].
- Addressed issue where updates to documents with text-indexed fields could lead to incorrect entries SERVER-14738[476].
- Resolved issue where creating descending index on `_id` could corrupt namespace SERVER-14833[477].
- *2.4.11 Changelog* (page 860).
- All 2.4.11 improvements[478].

2.4.10 – April 4, 2014

- Performs fast file allocation on Windows when available SERVER-8480[479].
- Start elections if more than one primary is detected SERVER-10793[480].
- Changes to allow safe downgrading from v2.6 to v2.4 SERVER-12914[481], SERVER-12175[482].
- Fixes for edge cases in index creation SERVER-12481[483], SERVER-12956[484].
- *2.4.10 Changelog* (page 860).
- All 2.4.10 improvements[485].

[470] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.13%22%20AND%20project%20%3D%20SERVER
[471] https://jira.mongodb.org/browse/SERVER-15111
[472] https://jira.mongodb.org/browse/SERVER-15369
[473] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.12%22%20AND%20project%20%3D%20SERVER
[474] https://jira.mongodb.org/browse/SERVER-14268
[475] https://jira.mongodb.org/browse/SERVER-12209
[476] https://jira.mongodb.org/browse/SERVER-14738
[477] https://jira.mongodb.org/browse/SERVER-14833
[478] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.11%22%20AND%20project%20%3D%20SERVER
[479] https://jira.mongodb.org/browse/SERVER-8480
[480] https://jira.mongodb.org/browse/SERVER-10793
[481] https://jira.mongodb.org/browse/SERVER-12914
[482] https://jira.mongodb.org/browse/SERVER-12175
[483] https://jira.mongodb.org/browse/SERVER-12481
[484] https://jira.mongodb.org/browse/SERVER-12956
[485] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.10%22%20AND%20project%20%3D%20SERVER

2.4.9 – January 10, 2014

- Fix for instances where `mongos` incorrectly reports a successful write SERVER-12146[486].
- Make non-primary read preferences consistent with `slaveOK` versioning logic SERVER-11971[487].
- Allow new sharded cluster connections to read from secondaries when primary is down SERVER-7246[488].
- All 2.4.9 improvements[489].

2.4.8 – November 1, 2013

- Increase future compatibility for 2.6 authorization features SERVER-11478[490].
- Fix `dbhash` cache issue for config servers SERVER-11421[491].
- All 2.4.8 improvements[492].

2.4.7 – October 21, 2013

- Fixed over-aggressive caching of V8 Isolates SERVER-10596[493].
- Removed extraneous initial count during mapReduce SERVER-9907[494].
- Cache results of dbhash command SERVER-11021[495].
- Fixed memory leak in aggregation SERVER-10554[496].
- All 2.4.7 improvements[497].

2.4.6 – August 20, 2013

- Fix for possible loss of documents during the chunk migration process if a document in the chunk is very large SERVER-10478[498].
- Fix for C++ client shutdown issues SERVER-8891[499].
- Improved replication robustness in presence of high network latency SERVER-10085[500].
- Improved Solaris support SERVER-9832[501], SERVER-9786[502], and SERVER-7080[503].

[486] https://jira.mongodb.org/browse/SERVER-12146
[487] https://jira.mongodb.org/browse/SERVER-11971
[488] https://jira.mongodb.org/browse/SERVER-7246
[489] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
[490] https://jira.mongodb.org/browse/SERVER-11478
[491] https://jira.mongodb.org/browse/SERVER-11421
[492] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
[493] https://jira.mongodb.org/browse/SERVER-10596
[494] https://jira.mongodb.org/browse/SERVER-9907
[495] https://jira.mongodb.org/browse/SERVER-11021
[496] https://jira.mongodb.org/browse/SERVER-10554
[497] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
[498] https://jira.mongodb.org/browse/SERVER-10478
[499] https://jira.mongodb.org/browse/SERVER-8891
[500] https://jira.mongodb.org/browse/SERVER-10085
[501] https://jira.mongodb.org/browse/SERVER-9832
[502] https://jira.mongodb.org/browse/SERVER-9786
[503] https://jira.mongodb.org/browse/SERVER-7080

- All 2.4.6 improvements[504].

2.4.5 – July 3, 2013

- Fix for CVE-2013-4650 Improperly grant user system privileges on databases other than local SERVER-9983[505].
- Fix for CVE-2013-3969 Remotely triggered segmentation fault in Javascript engine SERVER-9878[506].
- Fix to prevent identical background indexes from being built SERVER-9856[507].
- Config server performance improvements SERVER-9864[508] and SERVER-5442[509].
- Improved initial sync resilience to network failure SERVER-9853[510].
- All 2.4.5 improvements[511].

2.4.4 – June 4, 2013

- Performance fix for Windows version SERVER-9721[512]
- Fix for config upgrade failure SERVER-9661[513].
- Migration to Cyrus SASL library for MongoDB Enterprise SERVER-8813[514].
- All 2.4.4 improvements[515].

2.4.3 – April 23, 2013

- Fix for `mongo` shell ignoring modified object's `_id` field SERVER-9385[516].
- Fix for race condition in log rotation SERVER-4739[517].
- Fix for `copydb` command with authorization in a sharded cluster SERVER-9093[518].
- All 2.4.3 improvements[519].

2.4.2 – April 17, 2013

- Several V8 memory leak and performance fixes SERVER-9267[520] and SERVER-9230[521].

[504] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
[505] https://jira.mongodb.org/browse/SERVER-9983
[506] https://jira.mongodb.org/browse/SERVER-9878
[507] https://jira.mongodb.org/browse/SERVER-9856
[508] https://jira.mongodb.org/browse/SERVER-9864
[509] https://jira.mongodb.org/browse/SERVER-5442
[510] https://jira.mongodb.org/browse/SERVER-9853
[511] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
[512] https://jira.mongodb.org/browse/SERVER-9721
[513] https://jira.mongodb.org/browse/SERVER-9661
[514] https://jira.mongodb.org/browse/SERVER-8813
[515] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
[516] https://jira.mongodb.org/browse/SERVER-9385
[517] https://jira.mongodb.org/browse/SERVER-4739
[518] https://jira.mongodb.org/browse/SERVER-9093
[519] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
[520] https://jira.mongodb.org/browse/SERVER-9267
[521] https://jira.mongodb.org/browse/SERVER-9230

- Fix for upgrading sharded clusters SERVER-9125[522].
- Fix for high volume connection crash SERVER-9014[523].
- All 2.4.2 improvements[524]

2.4.1 – April 17, 2013

- Fix for losing index changes during initial sync SERVER-9087[525]
- All 2.4.1 improvements[526].

Major New Features

The following changes in MongoDB affect both standard and Enterprise editions:

Text Search

Add support for text search of content in MongoDB databases as a *beta* feature. See *Text Indexes* (page 501) for more information.

Geospatial Support Enhancements

- Add new *2dsphere index* (page 497). The new index supports GeoJSON[527] objects `Point`, `LineString`, and `Polygon`. See *2dsphere Indexes* (page 497) and *Geospatial Indexes and Queries* (page 494).
- Introduce operators `$geometry`, `$geoWithin` and `$geoIntersects` to work with the GeoJSON data.

Hashed Index

Add new *hashed index* (page 504) to index documents using hashes of field values. When used to index a shard key, the hashed index ensures an evenly distributed shard key. See also *Hashed Shard Keys* (page 689).

Improvements to the Aggregation Framework

- Improve support for geospatial queries. See the `$geoWithin` operator and the `$geoNear` pipeline stage.
- Improve sort efficiency when the `$sort` stage immediately precedes a `$limit` in the pipeline.
- Add new operators `$millisecond` and `$concat` and modify how `$min` operator processes `null` values.

[522] https://jira.mongodb.org/browse/SERVER-9125
[523] https://jira.mongodb.org/browse/SERVER-9014
[524] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
[525] https://jira.mongodb.org/browse/SERVER-9087
[526] https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
[527] http://geojson.org/geojson-spec.html

Changes to Update Operators

- Add new `$setOnInsert` operator for use with `upsert` .
- Enhance functionality of the `$push` operator, supporting its use with the `$each`, the `$sort`, and the `$slice` modifiers.

Additional Limitations for Map-Reduce and `$where` Operations

The `mapReduce` command, `group` command, and the `$where` operator expressions cannot access certain global functions or properties, such as `db`, that are available in the `mongo` shell. See the individual command or operator for details.

Improvements to `serverStatus` Command

Provide additional metrics and customization for the `serverStatus` command. See `db.serverStatus()` and `serverStatus` for more information.

Security Enhancements

- Introduce a role-based access control system User Privileges[528] now use a new format for `Privilege Documents`.
- Enforce uniqueness of the user in user privilege documents per database. Previous versions of MongoDB did not enforce this requirement, and existing databases may have duplicates.
- Support encrypted connections using SSL certificates signed by a Certificate Authority. See *Configure mongod and mongos for TLS/SSL* (page 338).

For more information on security and risk management strategies, see *MongoDB Security Practices and Procedures* (page 313).

Performance Improvements

V8 JavaScript Engine

	On this page
JavaScript Changes in MongoDB 2.4	• Improved Concurrency (page 868) • Modernized JavaScript Implementation (ES5) (page 868) • Removed Non-Standard SpiderMonkey Features (page 868)

Consider the following impacts of *V8 JavaScript Engine* (page 867) in MongoDB 2.4:

Tip

Use the new `interpreterVersion()` method in the `mongo` shell and the `javascriptEngine` field in the output of `db.serverBuildInfo()` to determine which JavaScript engine a MongoDB binary uses.

[528] http://docs.mongodb.org/v2.4/reference/user-privileges

Improved Concurrency Previously, MongoDB operations that required the JavaScript interpreter had to acquire a lock, and a single `mongod` could only run a single JavaScript operation at a time. The switch to V8 improves concurrency by permitting multiple JavaScript operations to run at the same time.

Modernized JavaScript Implementation (ES5) The 5th edition of ECMAscript[529], abbreviated as ES5, adds many new language features, including:

- standardized JSON[530],
- strict mode[531],
- function.bind()[532],
- array extensions[533], 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[534], SERVER-8223[535], SERVER-8215[536], and SERVER-8214[537] 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[538] JavaScript extensions, previously supported by MongoDB's use of SpiderMonkey as its JavaScript engine.

E4X Extensions V8 does not support the *non-standard* E4X[539] extensions. E4X provides a native XML[540] 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()`

[529] http://www.ecma-international.org/publications/standards/Ecma-262.htm
[530] http://www.ecma-international.org/ecma-262/5.1/#sec-15.12.1
[531] http://www.ecma-international.org/ecma-262/5.1/#sec-4.2.2
[532] http://www.ecma-international.org/ecma-262/5.1/#sec-15.3.4.5
[533] http://www.ecma-international.org/ecma-262/5.1/#sec-15.4.4.16
[534] https://jira.mongodb.org/browse/SERVER-8216
[535] https://jira.mongodb.org/browse/SERVER-8223
[536] https://jira.mongodb.org/browse/SERVER-8215
[537] https://jira.mongodb.org/browse/SERVER-8214
[538] https://developer.mozilla.org/en-US/docs/SpiderMonkey
[539] https://developer.mozilla.org/en-US/docs/E4X
[540] https://developer.mozilla.org/en-US/docs/E4X/Processing_XML_with_E4X

- `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[541]

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[542].

`InternalError()` V8 does not support `InternalError()`. Use `Error()` instead.

`for each...in` Construct V8 does not support the use of for each...in[543] 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);
});
```

---

[541] https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_(Merge_into_own_page.2Fsection)
[542] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Iterators_and_Generators
[543] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in

Array Comprehension V8 does not support Array comprehensions[544].

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 872) 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[545].

Example

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```
function test () {
   if (false) {
```

[544] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Predefined_Core_Objects#Array_comprehensions
[545] https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Functions

```
      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[546].

```
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[547]. 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);
```

[546] http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/
[547] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String#String_generic_methods

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[548]. 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()[549]. Use the `Array` instance method `toString()` instead.

`uneval()` V8 does not support the non-standard method `uneval()`. Use the standardized JSON.stringify()[550] method instead.

Change default JavaScript engine from SpiderMonkey to V8. The change provides improved concurrency for JavaScript operations, modernized JavaScript implementation, and the removal of non-standard SpiderMonkey features, and affects all JavaScript behavior including the commands `mapReduce`, `group`, and `eval` and the query operator `$where`.

[548] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array#Array_generic_methods
[549] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/toSource
[550] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/JSON/stringify

See *JavaScript Changes in MongoDB 2.4* (page 867) for more information about all changes .

BSON Document Validation Enabled by Default for `mongod` and `mongorestore`

Enable basic *BSON* object validation for `mongod` and `mongorestore` when writing to MongoDB data files. See `wireObjectCheck` for details.

Index Build Enhancements

- Add support for multiple concurrent index builds in the background by a single `mongod` instance. See *building indexes in the background* (page 510) for more information on background index builds.
- Allow the `db.killOp()` method to terminate a foreground index build.
- Improve index validation during index creation. See *Compatibility and Index Type Changes in MongoDB 2.4* (page 881) for more information.

Set Parameters as Command Line Options

Provide `--setParameter` as a command line option for `mongos` and `mongod`. See `mongod` and `mongos` for list of available options for `setParameter`.

Changed Replication Behavior for Chunk Migration

By default, each document move during *chunk migration* (page 700) in a *sharded cluster* propagates to at least one secondary before the balancer proceeds with its next operation. See *Chunk Migration and Replication* (page 701).

Improved Chunk Migration Queue Behavior

Increase performance for moving multiple chunks off an overloaded shard. The balancer no longer waits for the current migration's delete phase to complete before starting the next chunk migration. See *Chunk Migration Queuing* (page 701) for details.

Enterprise

The following changes are specific to MongoDB Enterprise Editions:

SASL Library Change

In 2.4.4, MongoDB Enterprise uses Cyrus SASL. Earlier 2.4 Enterprise versions use GNU SASL (`libgsasl`). To upgrade to 2.4.4 MongoDB Enterprise or greater, you **must** install all package dependencies related to this change, including the appropriate Cyrus SASL `GSSAPI` library. See *Install MongoDB Enterprise* (page 27) for details of the dependencies.

New Modular Authentication System with Support for Kerberos

In 2.4, the MongoDB Enterprise now supports authentication via a Kerberos mechanism. See *Configure MongoDB with Kerberos Authentication on Linux* (page 369) for more information. For drivers that provide support for Kerberos authentication to MongoDB, refer to *Driver Support* (page 329).

For more information on security and risk management strategies, see *MongoDB Security Practices and Procedures* (page 313).

Additional Information

Platform Notes

For OS X, MongoDB 2.4 only supports OS X versions 10.6 (Snow Leopard) and later. There are no other platform support changes in MongoDB 2.4. See the downloads page[551] for more information on platform support.

Upgrade Process

Upgrade MongoDB to 2.4

On this page

In the general case, the upgrade from MongoDB 2.2 to 2.4 is a binary-compatible "drop-in" upgrade: shut down the `mongod` instances and replace them with `mongod` instances running 2.4. **However**, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for *upgrading sharded clusters* (page 875) and the considerations for *reverting to 2.2 after running 2.4* (page 879).

Upgrade Recommendations and Checklist When upgrading, consider the following:

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` instance or instances.
- To upgrade to 2.4 sharded clusters *must* upgrade following the *meta-data upgrade procedure* (page 875).
- If you're using 2.2.0 and running with `authorization` 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 879).
- If you have `system.users` documents (i.e. for `authorization`) that you created before 2.4 you *must* ensure that there are no duplicate values for the `user` field in the `system.users` collection in *any* database. If you *do* have documents with duplicate user fields, you must remove them before upgrading.

  See *Security Enhancements* (page 867) for more information.

[551] http://www.mongodb.org/downloads/

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[552]. See *Install MongoDB* (page 5) for more information.
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.4 `mongod` binary and restart `mongod`.

Upgrade a Replica Set from MongoDB 2.2 to MongoDB 2.4 You can upgrade to 2.4 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 2.2 binary with the 2.4 binary. After upgrading a `mongod` instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.
2. Use the `mongo` shell method `rs.stepDown()` to step down the *primary* to allow the normal *failover* (page 583) procedure. `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

   Once the primary has stepped down and another member has assumed `PRIMARY` state, as observed in the output of `rs.status()`, shut down the previous primary and replace `mongod` binary with the 2.4 binary and start the new process.

Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

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.

Overview Upgrading a *sharded cluster* from MongoDB version 2.2 to 2.4 (or 2.3) requires that you run a 2.4 `mongos` with the `--upgrade` option, described in this procedure. The upgrade process does not require downtime.

The upgrade to MongoDB 2.4 adds epochs to the meta-data for all collections and chunks in the existing cluster. MongoDB 2.2 processes are capable of handling epochs, even though 2.2 did not require them. This procedure applies only to upgrades from version 2.2. Earlier versions of MongoDB do not correctly handle epochs. See *Cluster Meta-data Upgrade* (page 875) for more information.

After completing the meta-data upgrade you can fully upgrade the components of the cluster. With the balancer disabled:

- Upgrade all `mongos` instances in the cluster.
- Upgrade all 3 `mongod` config server instances.
- Upgrade the `mongod` instances for each shard, one at a time.

See *Upgrade Sharded Cluster Components* (page 879) for more information.

Cluster Meta-data Upgrade

[552] http://www.mongodb.org/downloads

Considerations Beware of the following properties of the cluster upgrade process:

- Before you start the upgrade, ensure that the amount of free space on the filesystem for the *config database* (page 754) is at least 4 to 5 times the amount of space currently used by the *config database* (page 754) data files.

  Additionally, ensure that all indexes in the *config database* (page 754) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

  The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.

- While the upgrade is in progress, you cannot make changes to the collection meta-data. For example, during the upgrade, do **not** perform:
  - `sh.enableSharding()`,
  - `sh.shardCollection()`,
  - `sh.addShard()`,
  - `db.createCollection()`,
  - `db.collection.drop()`,
  - `db.dropDatabase()`,
  - any operation that creates a database, or
  - any other operation that modifies the cluster meta-data in any way. See *Sharding Reference* (page 753) for a complete list of sharding commands. Note, however, that not all commands on the *Sharding Reference* (page 753) page modifies the cluster meta-data.
- Once you upgrade to 2.4 and complete the upgrade procedure **do not** use 2.0 `mongod` and `mongos` processes in your cluster. 2.0 process may re-introduce old meta-data formats into cluster meta-data.

The upgraded config database will require more storage space than before, to make backup and working copies of the `config.chunks` (page 756) and `config.collections` (page 757) collections. As always, if storage requirements increase, the `mongod` might need to pre-allocate additional data files. See *What tools can I use to investigate storage use in MongoDB?* (page 794) for more information.

Meta-data Upgrade Procedure Changes to the meta-data format for sharded clusters, stored in the *config database* (page 754), 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 875) 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 754) is at least 4 to 5 times the amount of space currently used by the *config database* (page 754) data files.

   Additionally, ensure that all indexes in the *config database* (page 754) are `{v:1}` indexes. If a critical index is a `{v:0}` index, chunk splits can fail due to known issues with the `{v:0}` format. `{v:0}` indexes are present on clusters created with MongoDB 2.0 or earlier.

   The duration of the metadata upgrade depends on the network latency between the node performing the upgrade and the three config servers. Ensure low latency between the upgrade process and the config servers.

   To check the version of your indexes, use `db.collection.getIndexes()`.

   If any index **on the config database** is `{v:0}`, you should rebuild those indexes by connecting to the `mongos` and either: rebuild all indexes using the `db.collection.reIndex()` method, or drop and rebuild specific

indexes using `db.collection.dropIndex()` and then `db.collection.ensureIndex()`. If you need to upgrade the `_id` index to `{v:1}` use `db.collection.reIndex()`.

You may have `{v:0}` indexes on other databases in the cluster.

2. Turn off the *balancer* (page 698) in the *sharded cluster*, as described in *Disable the Balancer* (page 732).

Optional

For additional security during the upgrade, you can make a backup of the config database using `mongodump` or other backup tools.

---

3. Ensure there are no version 2.0 `mongod` or `mongos` processes still active in the sharded cluster. The automated upgrade process checks for 2.0 processes, but network availability can prevent a definitive check. Wait 5 minutes after stopping or upgrading version 2.0 `mongos` processes to confirm that none are still active.

4. Start a single 2.4 `mongos` process with `configDB` pointing to the sharded cluster's *config servers* (page 684) and with the `--upgrade` option. The upgrade process happens before the process becomes a daemon (i.e. before `--fork`.)

   You can upgrade an existing `mongos` instance to 2.4 or you can start a new *mongos* instance that can reach all config servers if you need to avoid reconfiguring a production `mongos`.

   Start the `mongos` with a command that resembles the following:

   ```
   mongos --configdb <config servers> --upgrade
   ```

   Without the `--upgrade` option 2.4 `mongos` processes will fail to start until the upgrade process is complete.

   The upgrade will prevent any chunk moves or splits from occurring during the upgrade process. If there are very many sharded collections or there are stale locks held by other failed processes, acquiring the locks for all collections can take seconds or minutes. See the log for progress updates.

5. When the `mongos` process starts successfully, the upgrade is complete. If the `mongos` process fails to start, check the log for more information.

   If the `mongos` terminates or loses its connection to the config servers during the upgrade, you may always safely retry the upgrade.

   However, if the upgrade failed during the short critical section, the `mongos` will exit and report that the upgrade will require manual intervention. To continue the upgrade process, you must follow the *Resync after an Interruption of the Critical Section* (page 878) procedure.

Optional

If the `mongos` logs show the upgrade waiting for the upgrade lock, a previous upgrade process may still be active or may have ended abnormally. After 15 minutes of no remote activity `mongos` will force the upgrade lock. If you can verify that there are no running upgrade processes, you may connect to a 2.2 `mongos` process and force the lock manually:

```
mongo <mongos.example.net>

db.getMongo().getCollection("config.locks").findOne({ _id : "configUpgrade" })
```

If the process specified in the `process` field of this document is *verifiably* offline, run the following operation to force the lock.

```
db.getMongo().getCollection("config.locks").update({ _id : "configUpgrade" }, { $set : { state :
```

It is always more safe to wait for the `mongos` to verify that the lock is inactive, if you have any doubts about the activity of another upgrade operation. In addition to the `configUpgrade`, the `mongos` may need to wait for specific collection locks. Do not force the specific collection locks.

6. Upgrade and restart other `mongos` processes in the sharded cluster, *without* the `--upgrade` option.

   See *Upgrade Sharded Cluster Components* (page 879) for more information.

7. *Re-enable the balancer* (page 732). 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 684) must be re-synced, and there may be problems starting new `mongos` processes. The *sharded cluster* will remain accessible, but avoid all cluster meta-data changes until you resync the config servers. Operations that change meta-data include: adding shards, dropping databases, and dropping collections.

Note: Only perform the following procedure *if* something (e.g. network, power, etc.) interrupts the upgrade process during the short critical section of the upgrade. Remember, you may always safely attempt the *meta data upgrade procedure* (page 876).

To resync the config servers:

1. Turn off the *balancer* (page 698) 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 875)), you have already disabled the balancer.
2. Shut down two of the three config servers, preferably the last two listed in the `configDB` string. For example, if your `configDB` string is `configA:27019,configB:27019,configC:27019`, shut down `configB` and `configC`. Shutting down the last two config servers ensures that most `mongos` instances will have uninterrupted access to cluster meta-data.
3. `mongodump` the data files of the active config server (`configA`).
4. Move the data files of the deactivated config servers (`configB` and `configC`) to a backup location.
5. Create new, empty *data directories*.
6. Restart the disabled config servers with `--dbpath` pointing to the now-empty data directory and `--port` pointing to an alternate port (e.g. `27020`).
7. Use `mongorestore` to repopulate the data files on the disabled documents from the active config server (`configA`) to the restarted config servers on the new port (`configB:27020,configC:27020`). These config servers are now re-synced.
8. Restart the restored config servers on the old port, resetting the port back to the old settings (`configB:27019` and `configC:27019`).
9. In some cases connection pooling may cause spurious failures, as the `mongos` disables old connections only after attempted use. 2.4 fixes this problem, but to avoid this issue in version 2.2, you can restart all `mongos` instances (one-by-one, to avoid downtime) and use the `rs.stepDown()` method before restarting each of the shard *replica set primaries*.
10. The sharded cluster is now fully resynced; however before you attempt the upgrade process again, you must manually reset the upgrade state using a version 2.2 `mongos`. Begin by connecting to the 2.2 `mongos` with the `mongo` shell:

```
mongo <mongos.example.net>
```

Then, use the following operation to reset the upgrade process:

```
db.getMongo().getCollection("config.version").update({ _id : 1 }, { $unset : { upgradeState : 1
```

11. Finally retry the upgrade process, as in *Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4* (page 875).

Upgrade Sharded Cluster Components After you have successfully completed the meta-data upgrade process described in *Meta-data Upgrade Procedure* (page 876), and the 2.4 `mongos` instance starts, you can upgrade the other processes in your MongoDB deployment.

While the balancer is still disabled, upgrade the components of your sharded cluster in the following order:

- Upgrade all `mongos` instances in the cluster, in any order.
- Upgrade all 3 `mongod` config server instances, upgrading the *first* system in the *`mongos --configdb`* argument *last*.
- Upgrade each shard, one at a time, upgrading the `mongod` secondaries before running `replSetStepDown` and upgrading the primary of each shard.

When this process is complete, you can now *re-enable the balancer* (page 732).

Rolling Upgrade Limitation for 2.2.0 Deployments Running with `auth` Enabled MongoDB *cannot* support deployments that mix 2.2.0 and 2.4.0, or greater, components. MongoDB version 2.2.1 and later processes *can* exist in mixed deployments with 2.4-series processes. Therefore you cannot perform a rolling upgrade from MongoDB 2.2.0 to MongoDB 2.4.0. To upgrade a cluster with 2.2.0 components, use one of the following procedures.

1. Perform a rolling upgrade of all 2.2.0 processes to the latest 2.2-series release (e.g. 2.2.3) so that there are no processes in the deployment that predate 2.2.1. When there are no 2.2.0 processes in the deployment, perform a rolling upgrade to 2.4.0.
2. Stop all processes in the cluster. Upgrade all processes to a 2.4-series release of MongoDB, and start all processes at the same time.

Upgrade from 2.3 to 2.4 If you used a `mongod` from the 2.3 or 2.4-rc (release candidate) series, you can safely transition these databases to 2.4.0 or later; *however*, if you created `2dsphere` or `text` indexes using a `mongod` before v2.4-rc2, you will need to rebuild these indexes. For example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )

db.records.ensureIndex( { loc: "2dsphere" } )
db.records.ensureIndex( { records: "text" } )
```

Downgrade MongoDB from 2.4 to Previous Versions For some cases the on-disk format of data files used by 2.4 and 2.2 `mongod` is compatible, and you can upgrade and downgrade if needed. However, several new features in 2.4 are incompatible with previous versions:

- `2dsphere` indexes are incompatible with 2.2 and earlier `mongod` instances.
- `text` indexes are incompatible with 2.2 and earlier `mongod` instances.
- using a `hashed` index as a shard key are incompatible with 2.2 and earlier `mongos` instances.
- `hashed` indexes are incompatible with 2.0 and earlier `mongod` instances.

Important: Collections sharded using hashed shard keys, should **not** use 2.2 `mongod` instances, which cannot correctly support cluster operations for these collections.

If you completed the *meta-data upgrade for a sharded cluster* (page 875), 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 875), you cannot use 2.0 `mongod` or `mongos` instances in the same cluster.

If you complete the meta-data upgrade, you can safely downgrade components in any order. When upgrade again, always upgrade `mongos` instances before `mongod` instances.

Do not create `2dsphere` or `text` indexes in a cluster that has 2.2 components.

Considerations and Compatibility If you upgrade to MongoDB 2.4, and then need to run MongoDB 2.2 with the same data files, consider the following limitations.

- If you use a `hashed` index as the shard key index, which is only possible under 2.4 you will not be able to query data in this sharded collection. Furthermore, a 2.2 `mongos` cannot properly route an insert operation for a collections sharded using a `hashed` index for the shard key index: any data that you insert using a 2.2 `mongos`, will not arrive on the correct shard and will not be reachable by future queries.
- If you *never* create an `2dsphere` or `text` index, you can move between a 2.4 and 2.2 `mongod` for a given data set; however, after you create the first `2dsphere` or `text` index with a 2.4 `mongod` you will need to run a 2.2 `mongod` with the *`--upgrade`* option and drop any `2dsphere` or `text` index.

Upgrade and Downgrade Procedures

Basic Downgrade and Upgrade **Except** as described below, moving between 2.2 and 2.4 is a drop-in replacement:

- stop the existing `mongod`, using the *`--shutdown`* option as follows:

  ```
  mongod --dbpath /var/mongod/data --shutdown
  ```

  Replace `/var/mongod/data` with your MongoDB `dbPath`.

- start the new `mongod` processes with the same `dbPath` setting, for example:

  ```
  mongod --dbpath /var/mongod/data
  ```

  Replace `/var/mongod/data` with your MongoDB `dbPath`.

Downgrade to 2.2 After Creating a `2dsphere` or `text` Index If you have created `2dsphere` or `text` indexes while running a 2.4 `mongod` instance, you can downgrade at any time, by starting the `2.2` `mongod` with the *`--upgrade`* option as follows:

```
mongod --dbpath /var/mongod/data/ --upgrade
```

Then, you will need to drop any existing `2dsphere` or `text` indexes using `db.collection.dropIndex()`, for example:

```
db.records.dropIndex( { loc: "2dsphere" } )
db.records.dropIndex( "records_text" )
```

Warning: `--upgrade` will run `repairDatabase` on any database where you have created a `2dsphere` or `text` index, which will rebuild *all* indexes.

Troubleshooting Upgrade/Downgrade Operations If you do not use `--upgrade`, when you attempt to start a 2.2 `mongod` and you have created a `2dsphere` or `text` index, `mongod` will return the following message:

```
'need to upgrade database index_plugin_upgrade with pdfile version 4.6, new version: 4.5 Not upgradi
```

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 [553] 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.

Additional Resources

- MongoDB Major Version Upgrade Consulting Package[554]

Compatibility and Index Type Changes in MongoDB 2.4

On this page

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 879).

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 879) to downgrade these databases and run 2.2 if needed, however this will run a full database repair (as with `repairDatabase`) for all affected databases.

Index Type Validation In MongoDB 2.2 and earlier you could specify invalid index types that did not exist. In these situations, MongoDB would create an ascending (e.g. `1`) index. Invalid indexes include index types specified by strings that do not refer to an existing index type, and all numbers other than `1` and `-1`. [555]

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:

[553] The data file version (i.e. `pdfile version`) is independent and unrelated to the release version of MongoDB.

[554] https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade

[555] 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 874) for full upgrade instructions.

Other Resources

- MongoDB Downloads[556].
- All JIRA issues resolved in 2.4[557].
- All Backwards incompatible changes[558].
- All Third Party License Notices[559].

12.2.2 Release Notes for MongoDB 2.2

On this page

Upgrading

MongoDB 2.2 is a production release series and succeeds the 2.0 production release series.

MongoDB 2.0 data files are compatible with 2.2-series binaries without any special migration process. However, always perform the upgrade process for replica sets and sharded clusters using the procedures that follow.

Synopsis

- `mongod`, 2.2 is a drop-in replacement for 2.0 and 1.8.

[556] http://mongodb.org/downloads

[557] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.3.2%22,+%222.3.1%22,+%222
rc0%22,+%222.4.0-rc1%22,+%222.4.0-rc2%22,+%222.4.0-rc3%22%29

[558] https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(%222.3.2%22%2C%20%222.3.1%22%2C%20%222.3.0%2
rc0%22%2C%20%222.4.0-rc1%22%2C%20%222.4.0-rc2%22%2C%20%222.4.0-rc3%22)%20AND%20%22Backwards%20Compatibility%22%20in%20(%22Major%

[559] https://github.com/mongodb/mongo/blob/v2.4/distsrc/THIRD-PARTY-NOTICES

- Check your `driver` documentation for information regarding required compatibility upgrades, and always run the recent release of your driver.

  Typically, only users running with authentication, will need to upgrade drivers before continuing with the upgrade to 2.2.

- For all deployments using authentication, upgrade the drivers (i.e. client libraries), before upgrading the `mongod` instance or instances.

- For all upgrades of sharded clusters:

  – turn off the balancer during the upgrade process. See the *Disable the Balancer* (page 732) section for more information.

  – upgrade all `mongos` instances before upgrading any `mongod` instances.

Other than the above restrictions, 2.2 processes can interoperate with 2.0 and 1.8 tools and processes. You can safely upgrade the `mongod` and `mongos` components of a deployment one by one while the deployment is otherwise operational. Be sure to read the detailed upgrade procedures below before upgrading production systems.

Upgrading a Standalone `mongod`

1. Download binaries of the latest release in the 2.2 series from the MongoDB Download Page[560].
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.2 `mongod` binary and restart MongoDB.

Upgrading a Replica Set

You can upgrade to 2.2 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` in the `mongo` shell.
2. Use the `mongo` shell method `rs.stepDown()` to step down the *primary* to allow the normal *failover* (page 583) procedure. `rs.stepDown()` expedites the failover procedure and is preferable to shutting down the primary directly.

   Once the primary has stepped down and another member has assumed `PRIMARY` state, as observed in the output of `rs.status()`, shut down the previous primary and replace `mongod` binary with the 2.2 binary and start the new process.

   Note: Replica set failover is not instant but will render the set unavailable to read or accept writes until the failover process completes. Typically this takes 10 seconds or more. You may wish to plan the upgrade during a predefined maintenance window.

Upgrading a Sharded Cluster

Use the following procedure to upgrade a sharded cluster:

- *Disable the balancer* (page 732).

[560] http://downloads.mongodb.org/

- Upgrade all `mongos` instances *first*, in any order.
- Upgrade all of the `mongod` config server instances using the *stand alone* (page 883) 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 883) 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[561] for more information.

Changes

Major Features

Aggregation Framework The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` command exposes the aggregation framework, and the `aggregate()` helper in the `mongo` shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: *Aggregation Concepts* (page 439)
- Reference: *Aggregation Reference* (page 470)
- Examples: *Aggregation Examples* (page 453)

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 222) tutorial.

Concurrency Improvements MongoDB 2.2 increases the server's capacity for concurrent operations with the following improvements:

1. DB Level Locking[562]
2. Improved Yielding on Page Faults[563]
3. Improved Page Fault Detection on Windows[564]

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use. See *locks*, recordStats[565], `db.currentOp()`, `mongotop`, and `mongostat`.

Improved Data Center Awareness with Tag Aware Sharding MongoDB 2.2 adds additional support for geographic distribution or other custom partitioning for sharded collections in *clusters*. By using this "tag aware" sharding, you can automatically ensure that data in a sharded database system is always on specific shards. For example, with tag aware sharding, you can ensure that data is closest to the application servers that use that data most frequently.

[561] https://jira.mongodb.org/browse/SERVER-6902
[562] https://jira.mongodb.org/browse/SERVER-4328
[563] https://jira.mongodb.org/browse/SERVER-3357
[564] https://jira.mongodb.org/browse/SERVER-4538
[565] http://docs.mongodb.org/v2.2/reference/server-status

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* (page 591) and *write concern* (page 82). For example, shard tagging can pin all "USA" data to one or more logical shards, while replica set tagging can control which `mongod` instances (e.g. "`production`" or "`reporting`") the application uses to service requests.

See the documentation for the following helpers in the `mongo` shell that support tagged sharding configuration:

- `sh.addShardTag()`
- `sh.addTagRange()`
- `sh.removeShardTag()`

Also, see *Tag Aware Sharding* (page 746) and *Manage Shard Tags* (page 747).

Fully Supported Read Preference Semantics All MongoDB clients and drivers now support full *read preferences* (page 591), including consistent support for a full range of *read preference modes* (page 670) and *tag sets* (page 594). This support extends to the `mongos` and applies identically to single replica sets and to the replica sets for each shard in a *sharded cluster*.

Additional read preference support now exists in the `mongo` shell using the `readPref()` cursor method.

Compatibility Changes

Authentication Changes MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` instances.

If your cluster runs with authentication:

- For all drivers, use the latest release of your driver and check its release notes.
- In sharded environments, to ensure that your cluster remains available during the upgrade process you **must** use the *upgrade procedure for sharded clusters* (page 883).

`findAndModify` Returns Null Value for Upserts that Perform Inserts In version 2.2, for *upsert* that perform inserts with the `new` option set to `false`, `findAndModify` commands will now return the following output:

```
{ 'ok': 1.0, 'value': null }
```

In the `mongo` shell, upsert `findAndModify` operations that perform inserts (with `new` set to `false`.)only output a `null` value.

In version 2.0 these operations would return an empty document, e.g. `{ }`.

See: SERVER-6226[566] for more information.

`mongodump` 2.2 Output Incompatible with Pre-2.2 `mongorestore` If you use the `mongodump` tool from the 2.2 distribution to create a dump of a database, you must use a 2.2 (or later) version of `mongorestore` to restore that dump.

See: SERVER-6961[567] for more information.

[566] https://jira.mongodb.org/browse/SERVER-6226
[567] https://jira.mongodb.org/browse/SERVER-6961

`ObjectId().toString()` Returns String Literal `ObjectId("...")` In version 2.2, the `toString()` method returns the string representation of the *ObjectId()* (page 185) 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 185), which holds the hexadecimal string value in both versions.

`ObjectId().valueOf()` Returns hexadecimal string In version 2.2, the `valueOf()` method returns the value of the *ObjectId()* (page 185) 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 185) 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[568] and the *Are there any restrictions on the names of Collections?* (page 772) FAQ item.

[568] https://jira.mongodb.org/browse/SERVER-4442

Restrictions on Database Names for Windows Database names running on Windows can no longer contain the following characters:

```
/\. "*<>:|?
```

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, `mongod` will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584[569] and SERVER-6729[570] 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[571] for more information.

New `$elemMatch` Projection Operator The `$elemMatch` operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the `$elemMatch` reference and the SERVER-2238[572] and SERVER-828[573] 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[574] for more information.

Service Support for `mongos.exe` You may now run `mongos.exe` instances as a Windows Service. See the `mongos.exe` reference and *Configure a Windows Service for MongoDB* (page 25) and SERVER-1589[575] for more information.

Log Rotate Command Support MongoDB for Windows now supports log rotation by way of the `logRotate` database command. See SERVER-2612[576] for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency Labeled "2008+" on the Downloads Page[577], 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[578] for more information.

Tool Improvements

Index Definitions Handled by `mongodump` and `mongorestore` When you specify the *`--collection`* option to `mongodump`, `mongodump` will now backup the definitions for all indexes that exist on the source database. When

[569] https://jira.mongodb.org/browse/SERVER-4584
[570] https://jira.mongodb.org/browse/SERVER-6729
[571] https://jira.mongodb.org/browse/SERVER-5516
[572] https://jira.mongodb.org/browse/SERVER-2238
[573] https://jira.mongodb.org/browse/SERVER-828
[574] https://jira.mongodb.org/browse/SERVER-5648
[575] https://jira.mongodb.org/browse/SERVER-1589
[576] https://jira.mongodb.org/browse/SERVER-2612
[577] http://www.mongodb.org/downloads
[578] https://jira.mongodb.org/browse/SERVER-3844

you attempt to restore this backup with `mongorestore`, the target `mongod` will rebuild all indexes. See SERVER-808[579] for more information.

`mongorestore` now includes the `--noIndexRestore` option to provide the preceding behavior. Use `--noIndexRestore` to prevent `mongorestore` from building previous indexes.

`mongooplog` for Replaying Oplogs The `mongooplog` tool makes it possible to pull *oplog* entries from `mongod` instance and apply them to another `mongod` instance. You can use `mongooplog` to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873[580] case and the `mongooplog` reference.

Authentication Support for `mongotop` and `mongostat` `mongotop` and `mongostat` now contain support for username/password authentication. See SERVER-3875[581] and SERVER-3871[582] for more information regarding this change. Also consider the documentation of the following options for additional information:

- `mongotop --username`
- `mongotop --password`
- `mongostat --username`
- `mongostat --password`

Write Concern Support for `mongoimport` and `mongorestore` `mongoimport` now provides an option to halt the import if the operation encounters an error, such as a network interruption, a duplicate key exception, or a write error. The `--stopOnError` option will produce an error rather than silently continue importing data. See SERVER-3937[583] for more information.

In `mongorestore`, the `--w` option provides support for configurable write concern.

`mongodump` Support for Reading from Secondaries You can now run `mongodump` when connected to a *secondary* member of a *replica set*. See SERVER-3854[584] for more information.

`mongoimport` Support for full 16MB Documents Previously, `mongoimport` would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use `mongoimport` to import documents that are at least 16 megabytes ins size. See SERVER-4593[585] for more information.

`Timestamp()` Extended JSON format MongoDB extended JSON now includes a new `Timestamp()` type to represent the Timestamp type that MongoDB uses for timestamps in the *oplog* among other contexts.

This permits tools like `mongooplog` and `mongodump` to query for specific timestamps. Consider the following `mongodump` operation:

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483[586] for more information.

[579] https://jira.mongodb.org/browse/SERVER-808
[580] https://jira.mongodb.org/browse/SERVER-3873
[581] https://jira.mongodb.org/browse/SERVER-3875
[582] https://jira.mongodb.org/browse/SERVER-3871
[583] https://jira.mongodb.org/browse/SERVER-3937
[584] https://jira.mongodb.org/browse/SERVER-3854
[585] https://jira.mongodb.org/browse/SERVER-4593
[586] https://jira.mongodb.org/browse/SERVER-3483

Shell Improvements

Improved Shell User Interface 2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the `mongo` shell:

- Full Unicode support.
- Bash-like line editing features. See SERVER-4312[587] for more information.
- Multi-line command support in shell history. See SERVER-3470[588] for more information.
- Windows support for the `edit` command. See SERVER-3998[589] for more information.

Helper to load Server-Side Functions The `db.loadServerScripts()` loads the contents of the current database's `system.js` collection into the current `mongo` shell session. See SERVER-1651[590] for more information.

Support for Bulk Inserts If you pass an array of *documents* to the `insert()` method, the `mongo` shell will now perform a bulk insert operation. See SERVER-3819[591] and SERVER-2395[592] for more information.

Note: For bulk inserts on sharded clusters, the `getLastError` command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Operations

Support for Logging to Syslog See the SERVER-2957[593] case and the documentation of the `syslogFacility` run-time option or the *`mongod --syslog`* and *`mongos --syslog`* command line-options.

`touch` Command Added the `touch` command to read the data and/or indexes from a collection into memory. See: SERVER-2023[594] and `touch` for more information.

`indexCounters` No Longer Report Sampled Data `indexCounters` now report actual counters that reflect index use and state. In previous versions, these data were sampled. See SERVER-5784[595] and `indexCounters` for more information.

Padding Specifiable on `compact` Command See the documentation of the `compact` and the SERVER-4018[596] issue for more information.

[587] https://jira.mongodb.org/browse/SERVER-4312
[588] https://jira.mongodb.org/browse/SERVER-3470
[589] https://jira.mongodb.org/browse/SERVER-3998
[590] https://jira.mongodb.org/browse/SERVER-1651
[591] https://jira.mongodb.org/browse/SERVER-3819
[592] https://jira.mongodb.org/browse/SERVER-2395
[593] https://jira.mongodb.org/browse/SERVER-2957
[594] https://jira.mongodb.org/browse/SERVER-2023
[595] https://jira.mongodb.org/browse/SERVER-5784
[596] https://jira.mongodb.org/browse/SERVER-4018

Added Build Flag to Use System Libraries The Boost library, version 1.49, is now embedded in the MongoDB code base.

If you want to build MongoDB binaries using system Boost libraries, you can pass `scons` using the `--use-system-boost` flag, as follows:

```
scons --use-system-boost
```

When building MongoDB, you can also pass `scons` a flag to compile MongoDB using only system libraries rather than the included versions of the libraries. For example:

```
scons --use-system-all
```

See the SERVER-3829[597] and SERVER-5172[598] 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[599] and SERVER-4683[600]. For more information about TCMalloc, see the documentation of TCMalloc[601] itself.

Replication

Improved Logging for Replica Set Lag When *secondary* members of a replica set fall behind in replication, `mongod` now provides better reporting in the log. This makes it possible to track replication in general and identify what process may produce errors or halt replication. See SERVER-3575[602] for more information.

Replica Set Members can Sync from Specific Members The new `replSetSyncFrom` command and new `rs.syncFrom()` helper in the `mongo` shell make it possible for you to manually configure from which member of the set a replica will poll *oplog* entries. Use these commands to override the default selection logic if needed. Always exercise caution with `replSetSyncFrom` when overriding the default behavior.

Replica Set Members will not Sync from Members Without Indexes Unless `buildIndexes: false` To prevent inconsistency between members of replica sets, if the member of a replica set has `buildIndexes` (page 661) set to `true`, other members of the replica set will *not* sync from this member, unless they also have `buildIndexes` (page 661) set to `true`. See SERVER-4160[603] for more information.

New Option To Configure Index Pre-Fetching during Replication By default, when replicating options, *secondaries* will pre-fetch *Indexes* (page 481) associated with a query to improve replication throughput in most cases. The `replication.secondaryIndexPrefetch` setting and *`--replIndexPrefetch`* option allow administrators to disable this feature or allow the `mongod` to pre-fetch only the index on the `_id` field. See SERVER-6718[604] for more information.

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

[597] https://jira.mongodb.org/browse/SERVER-3829
[598] https://jira.mongodb.org/browse/SERVER-5172
[599] https://jira.mongodb.org/browse/SERVER-188
[600] https://jira.mongodb.org/browse/SERVER-4683
[601] http://goog-perftools.sourceforge.net/doc/tcmalloc.html
[602] https://jira.mongodb.org/browse/SERVER-3575
[603] https://jira.mongodb.org/browse/SERVER-4160
[604] https://jira.mongodb.org/browse/SERVER-6718

- Improved support for sharded MapReduce[605], and
- MapReduce will retry jobs following a config error[606].

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 703) documentation and SERVER-1506[607] for more information.

Migration Thresholds Modified The *migration thresholds* (page 699) 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 699) documentation for more information.

Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice[608] and the SERVER-4683[609] for more information.

Resources

- MongoDB Downloads[610].
- All JIRA issues resolved in 2.2[611].
- All backwards incompatible changes[612].
- All third party license notices[613].
- What's New in MongoDB 2.2 Online Conference[614].

12.2.3 Release Notes for MongoDB 2.0

On this page

[605] https://jira.mongodb.org/browse/SERVER-4521
[606] https://jira.mongodb.org/browse/SERVER-4158
[607] https://jira.mongodb.org/browse/SERVER-1506
[608] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES#L231
[609] https://jira.mongodb.org/browse/SERVER-4683
[610] http://mongodb.org/downloads
[611] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.1.0%22%2C+%222.1.1%22%2
rc0%22%2C+%222.2.0-rc1%22%2C+%222.2.0-rc2%22%29+ORDER+BY+component+ASC%2C+key+DESC
[612] https://jira.mongodb.org/issues/?filter=11225&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10483%2C%2010893%2C%2010894%2C%2
[613] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES
[614] http://www.mongodb.com/events/webinar/mongodb-online-conference-sept

Upgrading

Although the major version number has changed, MongoDB 2.0 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.8.

Preparation

Read through all release notes before upgrading, and ensure that no changes will affect your deployment.

If you create new indexes in 2.0, then downgrading to 1.8 is possible but you must reindex the new collections.

`mongoimport` and `mongoexport` now correctly adhere to the CSV spec for handling CSV input/output. This may break existing import/export workflows that relied on the previous behavior. For more information see SERVER-1097[615].

Journaling (page 309) is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the `--nojournal` run-time option. Otherwise, MongoDB creates journal files during startup. The first time you start `mongod` with journaling, you will see a delay as `mongod` creates new files. In addition, you may see reduced write throughput.

2.0 `mongod` instances are interoperable with 1.8 `mongod` instances; however, for best results, upgrade your deployments using the following procedures:

Upgrading a Standalone `mongod`

1. Download the v2.0.x binaries from the MongoDB Download Page[616].
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.0.x `mongod` binary and restart MongoDB.

Upgrading a Replica Set

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 1.8 binary with the 2.0.x binary from the MongoDB Download Page[617].
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 82) in your application code to confirm that each update reaches multiple servers.
3. Use the `rs.stepDown()` to step down the primary to allow the normal *failover* (page 583) procedure.

   `rs.stepDown()` and `replSetStepDown` provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

   When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` binary with the 2.0.x binary.

Upgrading a Sharded Cluster

1. Upgrade all *config server* instances *first*, in any order. Since config servers use two-phase commit, *shard* configuration metadata updates will halt until all are up and running.

[615] https://jira.mongodb.org/browse/SERVER-1097
[616] http://downloads.mongodb.org/
[617] http://downloads.mongodb.org/

2. Upgrade `mongos` routers in any order.

Changes

Compact Command

A `compact` command is now available for compacting a single collection and its indexes. Previously, the only way to compact was to repair the entire database.

Concurrency Improvements

When going to disk, the server will yield the write lock when writing data that is not likely to be in memory. The initial implementation of this feature now exists:

See SERVER-2563[618] 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 527). 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[619].
- The `repairDatabase` command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 893), invoke the `compact` command.

Once you create new indexes, downgrading to 1.8.x will require a re-index of any indexes created using 2.0. See *Build Old Style Indexes* (page 527).

[618] https://jira.mongodb.org/browse/SERVER-2563
[619] https://jira.mongodb.org/browse/SERVER-3866

Sharding Authentication

Applications can now use authentication with *sharded clusters*.

Replica Sets

Hidden Nodes in Sharded Clusters In 2.0, `mongos` instances can now determine when a member of a replica set becomes "hidden" without requiring a restart. In 1.8, `mongos` if you reconfigured a member as hidden, you *had* to restart `mongos` to prevent queries from reaching the hidden member.

Priorities Each *replica set* member can now have a priority value consisting of a floating-point from 0 to 1000, inclusive. Priorities let you control which member of the set you prefer to have as *primary* the member with the highest priority that can see a majority of the set will be elected primary.

For example, suppose you have a replica set with three members, `A`, `B`, and `C`, and suppose that their priorities are set as follows:

- `A`'s priority is `2`.
- `B`'s priority is `3`.
- `C`'s priority is `1`.

During normal operation, the set will always chose `B` as primary. If `B` becomes unavailable, the set will elect `A` as primary.

For more information, see the `priority` (page 662) 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 82) 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 *Data Center Awareness* (page 218).

Drivers may also support tag-aware reads. Instead of specifying `slaveOk`, you specify `slaveOk` with tags indicating which data-centers to read from. For details, see the Drivers[620] 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 82).

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 645).

[620] https://docs.mongodb.org/ecosystem/drivers

Primary Checks for a Caught up Secondary before Stepping Down To minimize time without a *primary*, the `rs.stepDown()` method will now fail if the primary does not see a *secondary* within 10 seconds of its latest optime. You can force the primary to step down anyway, but by default it will return an error message.

See also *Force a Member to Become Primary* (page 638).

Extended Shutdown on the Primary to Minimize Interruption When you call the `shutdown` command, the *primary* will refuse to shut down unless there is a *secondary* whose optime is within 10 seconds of the primary. If such a secondary isn't available, the primary will step down and wait up to a minute for the secondary to be fully caught up before shutting down.

Note that to get this behavior, you must issue the `shutdown` command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode When `repair` or `compact` runs on a *secondary*, the secondary will automatically drop into "recovering" mode until the operation finishes. This prevents clients from trying to read from it while it's busy.

Geospatial Features

Multi-Location Documents Indexing is now supported on documents which have multiple location objects, embedded either inline or in embedded 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 for 2d Indexes* (page 500).

Polygon searches Polygonal `$within` queries are also now supported for simple polygon shapes. For details, see the `$within` operator documentation.

Journaling Enhancements

- Journaling is now enabled by default for 64-bit platforms. Use the `--nojournal` command line option to disable it.
- The journal is now compressed for faster commits to disk.
- A new `--journalCommitInterval` run-time option exists for specifying your own group commit interval. The default settings do not change.
- A new `{ getLastError: { j: true } }` option is available to wait for the group commit. The group commit will happen sooner when a client is waiting on `{j: true}`. If journaling is disabled, `{j: true}` is a no-op.

New `ContinueOnError` Option for Bulk Insert

Set the `continueOnError` option for bulk inserts, in the `driver`, so that bulk insert will continue to insert any remaining documents even if an insert fails, as is the case with duplicate key exceptions or network interruptions. The `getLastError` command will report whether any inserts have failed, not just the last one. If multiple errors occur, the client will only receive the most recent `getLastError` results.

Note: For bulk inserts on sharded clusters, the `getLastError` command alone is insufficient to verify success. Applications should must verify the success of bulk inserts in application logic.

Map Reduce

Output to a Sharded Collection Using the new `sharded` flag, it is possible to send the result of a map/reduce to a sharded collection. Combined with the `reduce` or `merge` flags, it is possible to keep adding data to very large collections from map/reduce jobs.

For more information, see *Map-Reduce* (page 442) and the `mapReduce` reference.

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 the `mapReduce` reference.

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 `$regex`.

$and A special boolean `$and` query operator is now available.

Command Output Changes

The output of the `validate` command and the documents in the `system.profile` collection have both been enhanced to return information as BSON objects with keys for each value rather than as free-form strings.

Shell Features

Custom Prompt You can define a custom prompt for the `mongo` shell. You can change the prompt at any time by setting the prompt variable to a string or a custom JavaScript function returning a string. For examples, see *Use a Custom Prompt* (page 286).

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 the `mongo` reference.

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `authorization`) *all* database commands require authentication, *except* the following commands.

- `isMaster`
- `authenticate`

- `getnonce`
- `buildInfo`
- `ping`
- `isdbgrid`

Resources

- MongoDB Downloads[621]
- All JIRA Issues resolved in 2.0[622]
- All Backward Incompatible Changes[623]

12.2.4 Release Notes for MongoDB 1.8

On this page

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 898).
- The `mapReduce` command has changed in 1.8, causing incompatibility with previous releases. `mapReduce` no longer generates temporary collections (thus, `keepTemp` has been removed). Now, you must always supply a value for `out`. See the `out` field options in the `mapReduce` document. If you use MapReduce, this also likely means you need a recent version of your client driver.

Preparation

Read through all release notes before upgrading and ensure that no changes will affect your deployment.

Upgrading a Standalone `mongod`

1. Download the v1.8.x binaries from the MongoDB Download Page[624].
2. Shutdown your `mongod` instance.
3. Replace the existing binary with the 1.8.x `mongod` binary.
4. Restart MongoDB.

[621] http://mongodb.org/downloads
[622] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002
[623] https://jira.mongodb.org/issues/?filter=11023&jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20(10889%2C%2010886%2C%2010784%2C%2
[624] http://downloads.mongodb.org/

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[625].
2. Change your config (optional) to prevent election of a new primary.

   It is possible that, when you start shutting down members of the set, a new primary will be elected. To prevent this, you can give all of the secondaries a priority of `0` before upgrading, and then change them back afterwards. To do so:
   (a) Record your current config. Run `rs.config()` and paste the results into a text file.
   (b) Update your config so that all secondaries have priority `0`. For example:

```
config = rs.conf()
{
      "_id" : "foo",
      "version" : 3,
      "members" : [
              {
                      "_id" : 0,
                      "host" : "ubuntu:27017"
              },
              {
                      "_id" : 1,
                      "host" : "ubuntu:27018"
              },
              {
                      "_id" : 2,
                      "host" : "ubuntu:27019",
                      "arbiterOnly" : true
              }
              {
                      "_id" : 3,
                      "host" : "ubuntu:27020"
              },
              {
                      "_id" : 4,
                      "host" : "ubuntu:27021"
              },
      ]
}
config.version++
3
rs.isMaster()
{
      "setName" : "foo",
```

[625] http://downloads.mongodb.org/

```
        "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[626].
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[627].

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 898).
   - If the shard is a single `mongod` process, shut it down and then restart it with the 1.8.x binary from the MongoDB Download Page[628].
3. For each `mongos`:
   (a) Shut down the `mongos` process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[629].

[626] http://downloads.mongodb.org/
[627] http://downloads.mongodb.org/
[628] http://downloads.mongodb.org/
[629] http://downloads.mongodb.org/

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[630].
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 309) works fine, as journaling does not change anything about the data file format. Suppose you are running 1.8.x with journaling enabled and you decide to switch back to 1.6. There are two scenarios:

- If you shut down cleanly with 1.8.x, just restart with the 1.6 mongod binary.
- If 1.8.x shut down uncleanly, start 1.8.x up again and let the journal files run to fix any damage (incomplete writes) that may have existed at the crash. Then shut down 1.8.x cleanly and restart with the 1.6 mongod binary.

Changes

Journaling

MongoDB now supports write-ahead *Journaling Mechanics* (page 309) to facilitate fast crash recovery and durability in the storage engine. With journaling enabled, a `mongod` can be quickly restarted following a crash without needing to repair the *collections*. The aggregation framework makes it possible to do aggregation

Sparse and Covered Indexes

Sparse Indexes (page 507) 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 71) enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

Incremental MapReduce Support

The `mapReduce` command supports new options that enable incrementally updating existing *collections*. Previously, a MapReduce job could output either to a temporary collection or to a named permanent collection, which it would overwrite with new data.

You now have several options for the output of your MapReduce jobs:

[630] http://downloads.mongodb.org/

- You can merge MapReduce output into an existing collection. Output from the Reduce phase will replace existing keys in the output collection if it already exists. Other keys will remain in the collection.
- You can now re-reduce your output with the contents of an existing collection. Each key output by the reduce phase will be reduced with the existing document in the output collection.
- You can replace the existing output collection with the new results of the MapReduce job (equivalent to setting a permanent output collection in previous releases)
- You can compute MapReduce inline and return results to the caller without persisting the results of the job. This is similar to the temporary collections generated in previous releases, except results are limited to 8MB.

For more information, see the `out` field options in the `mapReduce` document.

Additional Changes and Enhancements

1.8.1

- Sharding migrate fix when moving larger chunks.
- Durability fix with background indexing.
- Fixed mongos concurrency issue with many incoming connections.

1.8.0

- All changes from 1.7.x series.

1.7.6

- Bug fixes.

1.7.5

- *Journaling* (page 309).
- Extent allocation improvements.
- Improved *replica set* connectivity for `mongos`.
- `getLastError` improvements for *sharding*.

1.7.4

- `mongos` routes `slaveOk` queries to *secondaries* in *replica sets*.
- New `mapReduce` output options.
- *Sparse Indexes* (page 507).

1.7.3

- Initial *covered index* (page 71) support.
- Distinct can use data from indexes when possible.
- `mapReduce` can merge or reduce results into an existing collection.
- `mongod` tracks and `mongostat` displays network usage. See *mongostat*.

- Sharding stability improvements.

1.7.2

- `$rename` operator allows renaming of fields in a document.
- `db.eval()` not to block.
- Geo queries with sharding.
- *`mongostat --discover`* option
- Chunk splitting enhancements.
- Replica sets network enhancements for servers behind a nat.

1.7.1

- Many sharding performance enhancements.
- Better support for `$elemMatch` on primitives in embedded arrays.
- Query optimizer enhancements on range queries.
- Window service enhancements.
- Replica set setup improvements.
- `$pull` works on primitives in arrays.

1.7.0

- Sharding performance improvements for heavy insert loads.
- Slave delay support for replica sets.
- `getLastErrorDefaults` (page 664) 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[631], 1.8.0[632]
- 1.7.6[633], 1.7.5[634], 1.7.4[635], 1.7.3[636], 1.7.2[637], 1.7.1[638], 1.7.0[639]

[631] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/v09MbhEm62Y
[632] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/JeHQOnam6Qk
[633] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/3t6GNZ1qGYc
[634] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/S5R0Tx9wkEg
[635] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/9Om3Vuw-y9c
[636] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/DfNUrdbmflI
[637] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/df7mwK6Xixo
[638] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/HUR9zYtTpA8
[639] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/TUnJCg9161A

Resources

- MongoDB Downloads[640]
- All JIRA Issues resolved in 1.8[641]

12.2.5 Release Notes for MongoDB 1.6

On this page

Upgrading

MongoDB 1.6 is a drop-in replacement for 1.4. To upgrade, simply shutdown `mongod` then restart with the new binaries.

Please note that you should upgrade to the latest version of whichever driver you're using. Certain drivers, including the Ruby driver, will require the upgrade, and all the drivers will provide extra features for connecting to replica sets.

Sharding

Sharding (page 675) is now production-ready, making MongoDB horizontally scalable, with no single point of failure. A single instance of `mongod` can now be upgraded to a distributed cluster with zero downtime when the need arises.

- *Sharding* (page 675)
- *Deploy a Sharded Cluster* (page 705)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 714)

Replica Sets

Replica sets (page 563), 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 563)
- *Deploy a Replica Set* (page 607)
- *Convert a Standalone to a Replica Set* (page 619)

[640] http://mongodb.org/downloads
[641] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10172

Other Improvements

- The `w` option (and `wtimeout`) forces writes to be propagated to `n` servers before returning success (this works especially well with replica sets)
- `$or queries`
- Improved concurrency
- `$slice` operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The `findAndModify` command now supports upserts. It also allows you to specify fields to return
- $showDiskLoc option to see disk location of a document
- Support for IPv6 and UNIX domain sockets

Installation

- Windows service improvements
- The C++ client is a separate tarball from the binaries

1.6.x Release Notes

- 1.6.5[642]

1.5.x Release Notes

- 1.5.8[643]
- 1.5.7[644]
- 1.5.6[645]
- 1.5.5[646]
- 1.5.4[647]
- 1.5.3[648]
- 1.5.2[649]
- 1.5.1[650]
- 1.5.0[651]

[642] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/06_QCC05Fpk
[643] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/uJfF1QN6Thk
[644] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/OYvz40RWs90
[645] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/4l0N2U_H0cQ
[646] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/oO749nvTARY
[647] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/380V_Ec_q1c
[648] https://groups.google.com/forum/?hl=en&fromgroups=#!topic/mongodb-user/hsUQL9CxTQw
[649] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/94EE3HVidAA
[650] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/7SBPQ2RSfdM
[651] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/VAhJcjDGTy0

You can see a full list of all changes on JIRA[652].

Thank you everyone for your support and suggestions!

12.2.6 Release Notes for MongoDB 1.4

On this page

Upgrading

We're pleased to announce the 1.4 release of MongoDB. 1.4 is a drop-in replacement for 1.2. To upgrade you just need to shutdown `mongod`, then restart with the new binaries. (Users upgrading from release 1.0 should review the *1.2 release notes* (page 906), in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

Core Server Enhancements

- *concurrency* (page 777) improvements
- indexing memory improvements
- *background index creation* (page 510)
- better detection of regular expressions so the index can be used in more cases

Replication and Sharding

- better handling for restarting slaves offline for a while
- fast new slaves from snapshots (`--fastsync`)
- configurable slave delay (`--slavedelay`)
- replication handles clock skew on master
- `$inc` replication fixes
- sharding alpha 3 - notably 2-phase commit on config servers

Deployment and Production

- *configure "slow threshold" for profiling* (page 240)
- ability to do `fsync + lock` for backing up raw files

[652] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107

- option for separate directory per db (`--directoryperdb`)
- `http://localhost:28017/_status` to get serverStatus via http
- REST interface is off by default for security (`--rest` to enable)
- can rotate logs with a db command, `logRotate`
- enhancements to `serverStatus` command (db.serverStatus()) - counters and *replication lag* (page 654) stats
- new `mongostat` tool

Query Language Improvements

- `$all` with regex
- `$not`
- partial matching of array elements `$elemMatch`
- $ operator for updating arrays
- `$addToSet`
- `$unset`
- `$pull` supports object matching
- `$set` with array indexes

Geo

- *2d geospatial search* (page 500)
- geo `$center` and `$box` searches

12.2.7 Release Notes for MongoDB 1.2.x

On this page

New Features

- More indexes per collection
- Faster index creation
- Map/Reduce
- Stored JavaScript functions
- Configurable fsync time
- Several small features and fixes

DB Upgrade Required

There are some changes that will require doing an upgrade if your previous version is <= 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` process
  - run `./mongod --upgrade`
  - start `mongod` again
- use a slave
  - start a slave on a different port and data directory
  - when its synced, shut down the master, and start the new slave on the regular port.

Ask in the forums or IRC for more help.

Replication Changes

- There have been minor changes in replication. If you are upgrading a master/slave setup from <= 1.1.2 you have to update the slave first.

mongoimport

- `mongoimportjson` has been removed and is replaced with `mongoimport` that can do json/csv/tsv

field filter changing

- We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter only changes the output, not which objects are returned. If you need that behavior, you can use `$exists`

12.3 Other MongoDB Release Notes

12.3.1 Default Write Concern Change

On this page

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.

   This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.

   The new top-level connection class is named `MongoClient`, or similar depending on how host languages handle namespacing.

2. The default write concern on the new `MongoClient` class will be to acknowledge all write operations [653]. This will allow your application to receive acknowledgment of all write operations.

   See the documentation of *Write Concern* (page 82) for more information about write concern in MongoDB.

   Please migrate to the new `MongoClient` class expeditiously.

Releases

The following driver releases will include the changes outlined in *Changes* (page 908). See each driver's release notes for a full account of each release as well as other related driver-specific changes.

- C#, version 1.7
- Java, version 2.10.0
- Node.js, version 1.2
- Perl, version 0.501.1
- PHP, version 1.4
- Python, version 2.4
- Ruby, version 1.8

12.4 MongoDB Version Numbers

For MongoDB `2.4.1`, `2.4` refers to the release series and `.1` refers to the revision. The second component of the release series (e.g. `4` in `2.4.1`) describes the type of release series. Release series ending with even numbers (e.g. `4` above) are *stable* and ready for production, while odd numbers are for *development* and testing only.

Generally, changes in the release series (e.g. `2.2` to `2.4`) mark the introduction of new features that may break backwards compatibility. Changes to the revision number mark the release bug fixes and backwards-compatible changes.

Important: Always upgrade to the latest stable revision of your release series.

The version numbering system for MongoDB differs from the system used for the MongoDB drivers. Drivers use only the first number to indicate a major version. For details, see *drivers-version-numbers*.

Example

[653] The drivers will call `getLastError` without arguments, which is logically equivalent to the `w: 1` option; however, this operation allows *replica set* users to override the default write concern with the `getLastErrorDefaults` (page 664) setting in the *Replica Set Configuration* (page 659).

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.

CHAPTER 13

About MongoDB Documentation

On this page

The MongoDB Manual[1] contains comprehensive documentation on MongoDB. 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.

13.1 License

13.2 Editions

In addition to the MongoDB Manual[3], you can also access this content in the following editions:

- PDF Format[4] (without reference).
- HTML tar.gz[5]
- ePub Format[6]

You also can access PDF files that contain subsets of the MongoDB Manual:

- MongoDB Reference Manual[7]

[1] http://docs.mongodb.org/manual/#
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/us/
[3] http://docs.mongodb.org/manual/#
[4] http://docs.mongodb.org/v2.6/MongoDB-manual.pdf
[5] http://docs.mongodb.org/v2.6/manual.tar.gz
[6] http://docs.mongodb.org/v2.6/MongoDB-manual.epub
[7] http://docs.mongodb.org/v2.6/MongoDB-reference-manual.pdf

- MongoDB CRUD Operations[8]
- Data Models for MongoDB[9]
- MongoDB Data Aggregation[10]
- Replication and MongoDB[11]
- Sharding and MongoDB[12]
- MongoDB Administration[13]
- MongoDB Security[14]

MongoDB Reference documentation is also available as part of dash[15]. You can also access the MongoDB Man Pages[16] which are also distributed with the official MongoDB Packages.

13.3 Version and Revisions

This version of the manual reflects version 2.6 of MongoDB.

See the MongoDB Documentation Project Page[17] 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[18].

This edition reflects "`v2.6`" branch of the documentation as of the "`a82057ac226cac8dee6b53a2ff6212b03abf817a`" revision. This branch is explicitly accessible via "http://docs.mongodb.org/v2.6" and you can always reference the commit of the current manual in the release.txt[19] file.

The most up-to-date, current, and stable version of the manual is always available at "http://docs.mongodb.org/manual/".

13.4 Report an Issue or Make a Change Request

To report an issue with this manual or to make a change request, file a ticket at the MongoDB DOCS Project on Jira[20].

13.5 Contribute to the Documentation

13.5.1 MongoDB Manual Translation

The original language of all MongoDB documentation is American English. However it is of critical importance to the documentation project to ensure that speakers of other languages can read and understand the documentation.

[8] http://docs.mongodb.org/v2.6/MongoDB-crud-guide.pdf
[9] http://docs.mongodb.org/v2.6/MongoDB-data-models-guide.pdf
[10] http://docs.mongodb.org/v2.6/MongoDB-aggregation-guide.pdf
[11] http://docs.mongodb.org/v2.6/MongoDB-replication-guide.pdf
[12] http://docs.mongodb.org/v2.6/MongoDB-sharding-guide.pdf
[13] http://docs.mongodb.org/v2.6/MongoDB-administration-guide.pdf
[14] http://docs.mongodb.org/v2.6/MongoDB-security-guide.pdf
[15] http://kapeli.com/dash
[16] http://docs.mongodb.org/v2.6/manpages.tar.gz
[17] http://docs.mongodb.org
[18] https://github.com/mongodb/docs
[19] http://docs.mongodb.org/v2.6/release.txt
[20] https://jira.mongodb.org/browse/DOCS

To this end, the MongoDB Documentation Project is preparing to launch a translation effort to allow the community to help bring the documentation to speakers of other languages.

If you would like to express interest in helping to translate the MongoDB documentation once this project is opened to the public, please:

- complete the MongoDB Contributor Agreement[21], and
- join the mongodb-translators[22] user group.

The mongodb-translators[23] user group exists to facilitate collaboration between translators and the documentation team at large. You can join the group without signing the Contributor Agreement, but you will not be allowed to contribute translations.

See also:

- *Contribute to the Documentation* (page 912)
- *Style Guide and Documentation Conventions* (page 914)
- *MongoDB Manual Organization* (page 923)
- *MongoDB Documentation Practices and Processes* (page 920)
- *MongoDB Documentation Build System* (page 924)

The entire documentation source for this manual is available in the mongodb/docs repository[24], which is one of the MongoDB project repositories on GitHub[25].

To contribute to the documentation, you can open a GitHub account[26], fork the mongodb/docs repository[27], make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the MongoDB Contributor Agreement[28].

You can clone the repository by issuing the following command at your system shell:

```
git clone git://github.com/mongodb/docs.git
```

13.5.2 About the Documentation Process

The MongoDB Manual uses Sphinx[29], a sophisticated documentation engine built upon Python Docutils[30]. The original reStructured Text[31] 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:

[21] http://www.mongodb.com/legal/contributor-agreement
[22] http://groups.google.com/group/mongodb-translators
[23] http://groups.google.com/group/mongodb-translators
[24] https://github.com/mongodb/docs
[25] http://github.com/mongodb
[26] https://github.com/
[27] https://github.com/mongodb/docs
[28] http://www.mongodb.com/contributor
[29] http://sphinx-doc.org//
[30] http://docutils.sourceforge.net/
[31] http://docutils.sourceforge.net/rst.html

Style Guide and Documentation Conventions

This document provides an overview of the style for the MongoDB documentation stored in this repository. The overarching goal of this style guide is to provide an accessible base style to ensure that our documentation is easy to read, simple to use, and straightforward to maintain.

For information regarding the MongoDB Manual organization, see *MongoDB Manual Organization* (page 923).

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.

2013-11-15: Added new table of preferred terms.

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/manual/core/sharding.rst` and `http://docs.mongodb.org/manual/administration/sharding.rst`.
  - For tutorials, the full title of the document should be in the file name. For example, `http://docs.mongodb.org/manual/tutorial/replace-one-configuration-server-in-a-sharc`
- Phrase headlines and titles so users can determine what questions the text will answer, and material that will be addressed, 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 through all documentation. For example, use "`replica-set-secondary-only-node`" as opposed to "`secondary-only-node`". This makes 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.

Headings 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 invalid 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 714) 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 714).

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

Preferred Term	Concept	Dispreferred Alternatives	Notes
document	A single, top-level object/record in a MongoDB collection.	record, object, row	Prefer document over object because of concerns about cross-driver language handling of objects. Reserve record for "allocation" of storage. Avoid "row," as possible.
database	A group of collections. Refers to a group of data files. This is the "logical" sense of the term "database."		Avoid genericizing "database." Avoid using database to refer to a server process or a data set. This applies both to the datastoring contexts as well as other (related) operational contexts (command context, authentication/authorization context.)
instance	A daemon process. (e.g. **mongos** or **mongod**)	process (acceptable sometimes), node (never acceptable), server.	Avoid using instance, unless it modifies something specifically. Having a descriptor for a process/instance makes it possible to avoid needing to make mongod or mongos plural. Server and node are both vague and contextually difficult to disambiguate with regards to application servers, and underlying hardware.
field name	The identifier of a value in a document.	key, column	Avoid introducing unrelated terms for a single field. In the documentation we've rarely had to discuss the identifier of a field, so the extra word here isn't burdensome.
field/value	The name/value pair that describes a unit of data in MongoDB.	key, slot, attribute	Use to emphasize the difference between the name of a field and its value For example, "_id" is the field and the default value is an ObjectId.
value	The data content of a field.	data	
MongoDB	A group of processes, or deployment that implement the MongoDB interface.	mongo, mongodb, cluster	Stylistic preference, mostly. In some cases it's useful to be able to refer generically to instances (that may be either **mongod** or **mongos**.)
embedded document	An embedded or nested document within a document or an array.	embedded document, nested document	
mapreduce	An operation performed by the mapReduce command.	mapReduce, map reduce, map/reduce	Avoid confusion with the command, shell helper, and driver interfaces. Makes it possible to discuss the operation generally.
cluster	A sharded cluster.	grid, shard cluster, set, deployment	Cluster is a great word for a group of processes; however, it's important to avoid letting the term become generic. Do not use for any group of MongoDB processes or deployments.
sharded cluster	A *sharded cluster*.	shard cluster, cluster, sharded system	
replica set	A deployment of replicating **mongod** programs that provide redundancy and automatic failover.	set, replication deployment	
deployment	A group of MongoDB processes, or a standalone **mongod** instance.	cluster, system	Typically in the form MongoDB deployment. Includes standalones, replica sets and sharded clusters.
data	The collection of physical	database, data	Important to keep the distinction between the

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".
- "embedded 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 embedded documents, in practice, users should only store coordinates in arrays. (See: DOCS-41[32].)

MongoDB Documentation Practices and Processes

This document provides an overview of the practices and processes.

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[33].

Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

[32] https://jira.mongodb.org/browse/DOCS-41
[33] http://jira.mongodb.org/

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[34] 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[35], fork the mongodb/docs repository[36], 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[37] 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.

[34] https://jira.mongodb.org/browse/DOCS
[35] https://github.com/
[36] https://github.com/mongodb/docs
[37] http://sphinx.pocoo.org/

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[38] project. These are better for more general changes that aren't necessarily tied to a specific line, or affect multiple files.

- Create a fork of the repository in your GitHub account, make any required changes and then create a pull request with your changes.

  If you insert lines that begin with any of the following annotations:

  ```
  .. TODO:
  TODO:
  .. TODO
  TODO
  ```

  followed by your comments, it will be easier for the original writer to locate your comments. The two dots `..` format is a comment in reStructured Text, which will hide your comments from Sphinx and publication if you're worried about that.

  This format is often easier for reviewers with larger portions of content to review.

MongoDB Manual Organization

This document provides an overview of the global organization of the documentation resource. Refer to the notes below if you are having trouble understanding the reasoning behind a file's current location, or if you want to add new documentation but aren't sure how to integrate it into the existing resource.

If you have questions, don't hesitate to open a ticket in the Documentation Jira Project[39] or contact the documentation team[40].

[38] http://jira.mongodb.org/browse/DOCS
[39] https://jira.mongodb.org/browse/DOCS
[40] docs@mongodb.com

Global Organization

Indexes and Experience The documentation project has two "index files": `http://docs.mongodb.org/manual/contents.txt` and `http://docs.mongodb.org/manual/index.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.

Topical Organization The placement of files in the repository depends on the *type* of documentation rather than the *topic* of the content. Like the difference between `contents.txt` and `index.txt`, by decoupling the organization of the files from the organization of the information the documentation can be more flexible and can more adequately address changes in the product and in users' needs.

Files in the `source/` directory represent the tip of a logical tree of documents, while *directories* are containers of types of content. The `administration` and `applications` directories, however, are legacy artifacts and with a few exceptions contain sub-navigation pages.

With several exceptions in the `reference/` directory, there is only one level of sub-directories in the `source/` directory.

Tools

The organization of the site, like all Sphinx sites derives from the `toctree` structure. However, in order to annotate the table of contents and provide additional flexibility, the MongoDB documentation generates `toctree` structures using data from YAML files stored in the `source/includes/` directory. These files start with `ref-toc` or `toc` and generate output in the `source/includes/toc/` directory. Briefly this system has the following behavior:

- files that start with `ref-toc` refer to the documentation of API objects (i.e. commands, operators and methods), and the build system generates files that hold `toctree` directives as well as files that hold *tables* that list objects and a brief description.
- files that start with `toc` refer to all other documentation and the build system generates files that hold `toctree` directives as well as files that hold *definition lists* that contain links to the documents and short descriptions the content.
- file names that have `spec` following `toc` or `ref-toc` will generate aggregated tables or definition lists and allow ad-hoc combinations of documents for landing pages and quick reference guides.

MongoDB Documentation Build System

This document contains more direct instructions for building the MongoDB documentation.

Getting Started

Install Dependencies The MongoDB Documentation project depends on the following tools:

- Python
- Git
- Inkscape (Image generation.)

- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Giza[41]

OS X Install Sphinx, Docutils, and their dependencies with `easy_install` the following command:

```
easy_install giza
```

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[42].

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[43]. This is **only** required to build PDFs.

Arch Linux Install packages from the system repositories with the following command:

```
pacman -S inkscape python2-pip
```

Then install the following Python packages:

```
pip2 install giza
```

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 inkscape python-pip
```

Then install the following Python packages:

```
pip install giza
```

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
```

[41] https://pypi.python.org/pypi/giza
[42] http://inkscape.org/download/
[43] http://www.tug.org/mactex/2011/

Building the Documentation

The MongoDB documentation build system is entirely accessible via `make` targets. For example, to build an HTML version of the documentation issue the following command:

```
make html
```

You can find the build output in `build/<branch>/html`, where `<branch>` is the name of the current branch.

In addition to the `html` target, the build system provides the following targets:

`publish` Builds and integrates all output for the production build. Build output is in `build/public/<branch>/`. When you run `publish` in the `master`, the build will generate some output in `build/public/`.

`push`; `stage` Uploads the production build to the production or staging web servers. Depends on `publish`. Requires access production or staging environment.

`push-all`; `stage-all` Uploads the entire content of `build/public/` to the web servers. Depends on `publish`. Not used in common practice.

`push-with-delete`; `stage-with-delete` Modifies the action of `push` and `stage` to remove remote file that don't exist in the local build. Use with caution.

`html`; `latex`; `dirhtml`; `epub`; `texinfo`; `man`; `json` These are standard targets derived from the default Sphinx Makefile, with adjusted dependencies. Additionally, for all of these targets you can append `-nitpick` to increase Sphinx's verbosity, or `-clean` to remove all Sphinx build artifacts.

`latex` performs several additional post-processing steps on `.tex` output generated by Sphinx. This target will also compile PDFs using `pdflatex`.

`html` and `man` also generates a `.tar.gz` file of the build outputs for inclusion in the final releases.

If you have any questions, please feel free to open a Jira Case[44].

[44] https://jira.mongodb.org/browse/DOCS