# Contents

1 Introduction to MongoDB 3  
  1.1 What is MongoDB 3

2 Install MongoDB 5  
  2.1 Recommended Operating Systems for Production Deployments 5  
  2.2 Other Supported Operating Systems 5  
  2.3 Installation Guides 5  
  2.4 First Steps with MongoDB 55  
  2.5 Additional Resources 55

3 MongoDB CRUD Operations 57  
  3.1 MongoDB CRUD Introduction 57  
  3.2 MongoDB CRUD Concepts 60  
  3.3 MongoDB CRUD Tutorials 92  
  3.4 MongoDB CRUD Reference 129

4 Data Models 143  
  4.1 Data Modeling Introduction 143  
  4.2 Data Modeling Concepts 145  
  4.3 Data Model Examples and Patterns 151  
  4.4 Data Model Reference 168

5 Administration 183  
  5.1 Administration Concepts 183  
  5.2 Administration Tutorials 220  
  5.3 Administration Reference 286  
  5.4 Production Checklist 311

6 Security 317  
  6.1 Security Introduction 317  
  6.2 Security Concepts 320  
  6.3 Security Tutorials 334  
  6.4 Security Reference 404  
  6.5 Security Checklist 431

7 Aggregation 435  
  7.1 Aggregation Introduction 435  
  7.2 Aggregation Concepts 439  
  7.3 Aggregation Examples 452
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>Aggregation Reference</td>
<td>469</td>
</tr>
<tr>
<td>7.5</td>
<td>Additional Resources</td>
<td>479</td>
</tr>
<tr>
<td>8</td>
<td>Indexes</td>
<td>481</td>
</tr>
<tr>
<td>8.1</td>
<td>Index Introduction</td>
<td>481</td>
</tr>
<tr>
<td>8.2</td>
<td>Index Concepts</td>
<td>486</td>
</tr>
<tr>
<td>8.3</td>
<td>Indexing Tutorials</td>
<td>517</td>
</tr>
<tr>
<td>8.4</td>
<td>Indexing Reference</td>
<td>552</td>
</tr>
<tr>
<td>9</td>
<td>Replication</td>
<td>559</td>
</tr>
<tr>
<td>9.1</td>
<td>Replication Introduction</td>
<td>559</td>
</tr>
<tr>
<td>9.2</td>
<td>Replication Concepts</td>
<td>563</td>
</tr>
<tr>
<td>9.3</td>
<td>Replica Set Tutorials</td>
<td>602</td>
</tr>
<tr>
<td>9.4</td>
<td>Replication Reference</td>
<td>651</td>
</tr>
<tr>
<td>10</td>
<td>Sharding</td>
<td>661</td>
</tr>
<tr>
<td>10.1</td>
<td>Sharding Introduction</td>
<td>661</td>
</tr>
<tr>
<td>10.2</td>
<td>Sharding Concepts</td>
<td>667</td>
</tr>
<tr>
<td>10.3</td>
<td>Sharded Cluster Tutorials</td>
<td>690</td>
</tr>
<tr>
<td>10.4</td>
<td>Sharding Reference</td>
<td>736</td>
</tr>
<tr>
<td>11</td>
<td>Frequently Asked Questions</td>
<td>743</td>
</tr>
<tr>
<td>11.1</td>
<td>FAQ: MongoDB Fundamentals</td>
<td>743</td>
</tr>
<tr>
<td>11.2</td>
<td>FAQ: MongoDB for Application Developers</td>
<td>746</td>
</tr>
<tr>
<td>11.3</td>
<td>FAQ: The <em>mongo</em> Shell</td>
<td>756</td>
</tr>
<tr>
<td>11.4</td>
<td>FAQ: Concurrency</td>
<td>758</td>
</tr>
<tr>
<td>11.5</td>
<td>FAQ: Sharding with MongoDB</td>
<td>763</td>
</tr>
<tr>
<td>11.6</td>
<td>FAQ: Replication and Replica Sets</td>
<td>768</td>
</tr>
<tr>
<td>11.7</td>
<td>FAQ: MongoDB Storage</td>
<td>772</td>
</tr>
<tr>
<td>11.8</td>
<td>FAQ: Indexes</td>
<td>777</td>
</tr>
<tr>
<td>11.9</td>
<td>FAQ: MongoDB Diagnostics</td>
<td>779</td>
</tr>
<tr>
<td>12</td>
<td>Release Notes</td>
<td>785</td>
</tr>
<tr>
<td>12.1</td>
<td>Current Stable Release</td>
<td>785</td>
</tr>
<tr>
<td>12.2</td>
<td>Previous Stable Releases</td>
<td>827</td>
</tr>
<tr>
<td>12.3</td>
<td>Other MongoDB Release Notes</td>
<td>924</td>
</tr>
<tr>
<td>12.4</td>
<td>MongoDB Version Numbers</td>
<td>924</td>
</tr>
<tr>
<td>13</td>
<td>About MongoDB Documentation</td>
<td>927</td>
</tr>
<tr>
<td>13.1</td>
<td>License</td>
<td>927</td>
</tr>
<tr>
<td>13.2</td>
<td>Editions</td>
<td>927</td>
</tr>
<tr>
<td>13.3</td>
<td>Version and Revisions</td>
<td>928</td>
</tr>
<tr>
<td>13.4</td>
<td>Report an Issue or Make a Change Request</td>
<td>928</td>
</tr>
<tr>
<td>13.5</td>
<td>Contribute to the Documentation</td>
<td>928</td>
</tr>
</tbody>
</table>
See About MongoDB Documentation (page 927) for more information about the MongoDB Documentation project, this Manual and additional editions of this text.

Note: This version of the PDF does not include the reference section, see MongoDB Reference Manual¹ for a PDF edition of all MongoDB Reference Material.

¹http://docs.mongodb.org/master/MongoDB-reference-manual.pdf
Introduction to MongoDB

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.

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

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 559) 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 661) distributes data across a cluster of machines.
- Replica sets can provide eventually-consistent reads for low-latency high throughput deployments.
Install MongoDB

MongoDB runs on most platforms and supports 64-bit architecture for production use and both 64-bit and 32-bit architectures for testing.

2.1 Recommended Operating Systems for Production Deployments

<table>
<thead>
<tr>
<th>Operating System</th>
<th>MongoDB</th>
<th>MongoDB Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Linux</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Debian 7.1</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>RedHat / CentOS 6.2+</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>SUSE 11</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Ubuntu LTS 12.04</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Ubuntu LTS 14.04</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>Windows Server 2012 &amp; 2012 R2</td>
<td>supported</td>
<td>supported</td>
</tr>
</tbody>
</table>

2.2 Other Supported Operating Systems

<table>
<thead>
<tr>
<th>Operating System</th>
<th>MongoDB</th>
<th>MongoDB Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac OSX 10.6+</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>RedHat / CentOS 5.5+</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>RedHat / CentOS 5.7+</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>RedHat / CentOS 7.0+</td>
<td>supported</td>
<td>supported</td>
</tr>
<tr>
<td>SmartOS</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Solaris 11 / SunOS 5.11 on x86</td>
<td>supported</td>
<td></td>
</tr>
<tr>
<td>Windows Server 2008 R2</td>
<td>supported</td>
<td>supported</td>
</tr>
</tbody>
</table>

2.3 Installation Guides

See the Release Notes (page 785) 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 7) Install MongoDB on Red Hat Enterprise and related Linux systems using .rpm packages.

Install on Amazon Linux (page 13) Install MongoDB on Amazon Linux systems using .rpm packages.
**Install on SUSE** (page 10) Install MongoDB on SUSE Linux systems using .rpm packages.

**Install on Ubuntu** (page 16) Install MongoDB on Ubuntu Linux systems using .deb packages.

**Install on Debian** (page 19) Install MongoDB on Debian systems using .deb packages.

**Install on Other Linux Systems** (page 22) Install the official build of MongoDB on other Linux systems from MongoDB archives.

**Install on OS X** (page 24) Install the official build of MongoDB on OS X systems from Homebrew packages or from MongoDB archives.

**Install on Windows** (page 27) Install MongoDB on Windows systems and optionally start MongoDB as a Windows service.

**Install MongoDB Enterprise** (page 33) 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 33) Install the MongoDB Enterprise build and required dependencies on Red Hat Enterprise or CentOS Systems using packages.

**Install MongoDB Enterprise on Ubuntu** (page 37) Install the MongoDB Enterprise build and required dependencies on Ubuntu Linux Systems using packages.

**Install MongoDB Enterprise on Amazon AMI** (page 47) Install the MongoDB Enterprise build and required dependencies on Amazon Linux AMI.

**Install MongoDB Enterprise on Windows** (page 49) Install the MongoDB Enterprise build and required dependencies using the .msi installer.

### 2.3.1 Install on Linux

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

**Recommended**

For the best installation experience, MongoDB provides packages for popular Linux distributions. These packages, which support specific platforms and provide improved performance and TLS/SSL support, are the preferred way to run MongoDB. The following guides detail the installation process for these systems:

**Install on Red Hat** (page 7) Install MongoDB on Red Hat Enterprise and related Linux systems using .rpm packages.

**Install on SUSE** (page 10) Install MongoDB on SUSE Linux systems using .rpm packages.

**Install on Amazon Linux** (page 13) Install MongoDB on Amazon Linux systems using .rpm packages.

**Install on Ubuntu** (page 16) Install MongoDB on Ubuntu Linux systems using .deb packages.

**Install on Debian** (page 19) 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, MongoDB provides a generic Linux release. These versions of MongoDB don’t include TLS/SSL, and may not perform as well as the targeted packages, but are compatible on most contemporary Linux systems. See the following guides for installation:
Install on Other Linux Systems (page 22) Install the official build of MongoDB on other Linux systems from MongoDB archives.

Install MongoDB on Red Hat Enterprise or CentOS Linux

Overview Use this tutorial to install MongoDB on Red Hat Enterprise Linux or CentOS Linux versions 5, 6, and 7 using .rpm packages. While some of these distributions include their own MongoDB packages, the official MongoDB packages are generally more up to date.

This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

Packages MongoDB provides officially supported packages in their own repository. This repository contains the following packages:

- mongodbc-rg
  This package is a metapackage that will automatically install the four component packages listed below.
- mongodbc-org-server
  This package contains the mongod daemon and associated configuration and init scripts.
- mongodbc-org-mongos
  This package contains the mongos daemon.
- mongodbc-org-shell
  This package contains the mongo shell.
- mongodbc-org-tools
  This package contains the following MongoDB tools: mongoimport bsodump, mongodump, mongoexport, mongoimport, mongooplog, mongoperf, mongorestore, mongostat, and mongotop.

Control Scripts The mongodbc-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 3.0.4, there are no control scripts for mongos. The mongos process is used only in sharding (page 667). 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 This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

The default /etc/mongod.conf configuration file supplied by the 3.0 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-org-3.0.repo file so that you can install MongoDB directly, using yum.

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

```
[mongodb-org-3.0]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/redhat/$releasever/mongodb-org/3.0/x86_64/
gpgcheck=0
enabled=1
```

Use the following repository to install only versions of MongoDB for the 3.0 release. If you’d like to install MongoDB packages from a particular release series (page 924), 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-org-2.6.repo file to hold the following configuration information for the MongoDB 2.6 repository:

```
[mongodb-org-2.6]
name=MongoDB 2.6 Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
gpgcheck=0
enabled=1
```

.repo files for each release can also be found in the repository itself\[^1\]. Remember that odd-numbered minor release versions (e.g. 2.5) are development versions and are unsuitable for production use.

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:

```
sudo yum install -y mongodb-org-3.0.4 mongodb-org-server-3.0.4 mongodb-org-shell-3.0.4 mongodb-org-mongos-3.0.4 mongodb-org-tools-3.0.4
```

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
```

Versions of the MongoDB packages before 2.6 use a different repo location. Refer to the version of the documentation appropriate for your MongoDB version.

Run MongoDB

Prerequisites

\[^1\]https://repo.mongodb.org/yum/redhat/
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.

- 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 prevent mongod from 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:

```bash
sudo chkconfig mongod on
```

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

```bash
sudo service mongod stop
```

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

```bash
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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See `getting-started` for the available editions.

Before deploying MongoDB in a production environment, consider the `Production Notes` (page 197) document.

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:

```bash
sudo service mongod stop
```

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

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

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

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

**Install MongoDB on SUSE**

**Overview**  Use this tutorial to install MongoDB on SUSE Linux from `.rpm` packages. While SUSE distributions include their own MongoDB packages, the official MongoDB packages are generally more up to date.

This installation guide only supports 64-bit systems. See `Platform Support` (page 809) for details.
**Packages**  MongoDB provides officially supported packages in their own repository. This repository contains 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, mongoimport, 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 3.0.4, there are no control scripts for mongos. The mongos process is used only in sharding (page 667). 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**  This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

The default /etc/mongod.conf configuration file supplied by the 3.0 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.

**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.

---

**Install MongoDB**

**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 latest stable release of MongoDB.

```
zypper addrepo --no-gpgcheck https://repo.mongodb.org/zypper/suse/11/mongodb-org/3.0/x86_64/ mongodb
```
This repository only offers the 3.0 MongoDB release. If you’d like to install MongoDB packages from a previous release series (page 924), 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, use the following command:

```
zypper addrepo --no-gpgcheck http://downloads-distro.mongodb.org/repo/suse/os/x86_64/mongodb
```

**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 zypper install 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:

```
sudo zypper install mongodb-org-3.0.4 mongodb-org-server-3.0.4 mongodb-org-shell-3.0.4 mongodb-org-mongos-3.0.4 mongodb-org-tools-3.0.4
```

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-org-3.0.4 mongodb-org-server-3.0.4 mongodb-org-shell-3.0.4 mongodb-org-mongos-3.0.4 mongodb-org-tools-3.0.4
```

Previous versions of MongoDB packages use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

**Run MongoDB** 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 help you start using MongoDB, MongoDB provides `Getting Started Guides` in various driver editions. See `getting-started` for the available editions.

Before deploying MongoDB in a production environment, consider the `Production Notes` (page 197) document.

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-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 Amazon Linux

**Overview**  Use this tutorial to install MongoDB on Amazon Linux from `.rpm` packages.

This installation guide only supports 64-bit systems. See `Platform Support` (page 809) for details.

**Packages**  MongoDB provides officially supported packages in their own repository. This repository contains 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*, *mongoimport*, *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 3.0.4, there are no control scripts for *mongos*. The *mongos* process is used only in *sharding* (page 667). You can use the *mongod* init script to derive your own *mongoose* control script for use in such environments. See the *mongos* reference for configuration details.

**Considerations** This installation guide only supports 64-bit systems. See **Platform Support** (page 809) for details.

The default `/etc/mongod.conf` configuration file supplied by the 3.0 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-org-3.0.repo` file so that you can install MongoDB directly, using **yum**.

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

```
[mongodb-org-3.0]
name=MongoDB Repository
baseurl=https://repo.mongodb.org/yum/amazon/2013.03/mongodb-org/3.0/x86_64/
gpgcheck=0
enabled=1
```

Use the following repository to install *only* versions of MongoDB for the 3.0 release. If you’d like to install MongoDB packages from a particular *release series* (page 924), 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-org-2.6.repo` file to hold the following configuration information for the MongoDB 2.6 repository:

```
[mongodb-org-2.6]
name=MongoDB 2.6 Repository
baseurl=http://downloads-distro.mongodb.org/repo/redhat/os/x86_64/
gpgcheck=0
enabled=1
```

*.repo* files for each release can also be found in the repository itself\(^2\). Remember that odd-numbered minor release versions (e.g. 2.5) are development versions and are unsuitable for production use.

\(^2\)https://repo.mongodb.org/yum/redhat/
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:

```
sudo yum install -y mongodb-org-3.0.4 mongodb-org-server-3.0.4 mongodb-org-shell-3.0.4 mongodb-org-mongos-3.0.4 mongodb-org-tools-3.0.4
```

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
```

Versions of the MongoDB packages before 2.6 use a different repo location. Refer to the version of the documentation appropriate for your MongoDB version.

Run MongoDB 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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

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

Platform Support

MongoDB only provides packages for 64-bit long-term support Ubuntu releases. Currently, this means 12.04 LTS (Precise Pangolin) and 14.04 LTS (Trusty Tahr). While the packages may work with other Ubuntu releases, this is not a supported configuration.

Packages MongoDB provides officially supported packages in their own repository. This repository contains 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.
• mongod-org-shell

This package contains the mongo shell.

• mongod-org-tools

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

Control Scripts  The mongdb-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 3.0.4, there are no control scripts for mongos. The mongos process is used only in sharding (page 667). 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  MongoDB only provides packages for 64-bit long-term support Ubuntu releases. Currently, this means 12.04 LTS (Precise Pangolin) and 14.04 LTS (Trusty Tahr). While the packages may work with other Ubuntu releases, this is not a supported configuration.

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

The default /etc/mongod.conf configuration file supplied by the 3.0 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 Key³:

```
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-org-3.0.list list file using the following command:

```
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

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.

³https://docs.mongodb.org/10gen-gpg-key.asc
Install the latest stable version of MongoDB. Issue the following command:

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

Install a specific release of MongoDB. To install a specific release, you must specify each component package individually along with the version number, as in the following example:

```bash
sudo apt-get install -y mongodb-org=3.0.4 mongodb-org-server=3.0.4 mongodb-org-shell=3.0.4 mongodb-org-mongos=3.0.4 mongodb-org-tools=3.0.4
```

If you only install `mongodb-org=3.0.4` and do not include the component packages, the latest version of each MongoDB package will be installed regardless of what version you specified.

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:

```bash
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
```

Versions of the MongoDB packages before 2.6 use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

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`:

```bash
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:

```bash
sudo service mongod stop
```

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

```bash
sudo service mongod restart
```
Step 5: Begin using MongoDB. To help you start using MongoDB, MongoDB provides *Getting Started Guides* in various driver editions. See *getting-started* for the available editions.

Before deploying MongoDB in a production environment, consider the *Production Notes* (page 197) document.

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:

```bash
sudo service mongod stop
```

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

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

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

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

Install MongoDB on Debian

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

MongoDB only provides packages for 64-bit Debian “Wheezy”. These packages may work with other Debian releases, but this is not a supported configuration.

Packages MongoDB provides officially supported packages in their own repository. This repository contains 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.
This package contains the following MongoDB tools: mongoimport, bsondump, mongodump, mongoexport, mongoimport, 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 3.0.4, there are no control scripts for mongos. The mongos process is used only in sharding (page 667). 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**  This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

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

The default `/etc/mongod.conf` configuration file supplied by the 3.0 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-org-3.0.list` file for MongoDB.**  Create the list file using the following command:

```
echo "deb http://repo.mongodb.org/apt/debian wheezy/mongodb-org/3.0 main" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

Currently packages are only available for Debian 7 (Wheezy).

**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.

\(^4\)https://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. To install a specific release, you must specify each component package individually along with the version number, as in the following example:

```
sudo apt-get install -y mongodb-org=3.0.4 mongodb-org-server=3.0.4 mongodb-org-shell=3.0.4 mongodb-org-mongos=3.0.4 mongodb-org-tools=3.0.4
```

If you only install `mongodb-org=3.0.4` and do not include the component packages, the latest version of each MongoDB package will be installed regardless of what version you specified.

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
```

Versions of the MongoDB packages before 2.6 use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

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
```
Step 5: Begin using MongoDB. To help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

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 https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-3.0.4.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_64-3.0.4.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_64-3.0.4/ 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](https://www.mongodb.org/downloads).

For example, to download the latest release through the shell, issue the following: .. include:: /includes/release/curl-release-linux-i686.rst

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-i686-3.0.4.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-i686-3.0.4/ 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:

dir -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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

2.3.2 Install MongoDB on OS X

Overview

Use this tutorial to install MongoDB on OS X systems.

Platform Support

Starting in version 3.0, MongoDB only supports OS X versions 10.7 (Lion) on Intel x86-64 and later.

MongoDB is available through the popular OS X package manager Homebrew\(^5\) or through the MongoDB Download site\(^6\).

\(^5\)http://brew.sh/

\(^6\)http://www.mongodb.org/downloads
Install MongoDB

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

Install MongoDB with Homebrew

Homebrew 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
```

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 25).

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 https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.4.tgz
```

---

7 http://brew.sh/
8 http://brew.sh/
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_64-3.0.4.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_64-3.0.4/ 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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

2.3.3 Install MongoDB on Windows

Overview

Use this tutorial to install MongoDB on 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 Windows9.

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:

9http://support.microsoft.com/kb/2731284
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.


To find which version of Windows you are running, enter the following commands in the Command Prompt or PowerShell:

```
wmic os get caption
wmic os get osarchitecture
```

**Step 2: Download MongoDB for Windows.**

Download the latest production release of MongoDB from the MongoDB downloads page\(^\text{10}\). 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. 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: Open an Administrator command prompt.**

Press the Win key, type cmd.exe, and press Ctrl + Shift + Enter to run the Command Prompt as Administrator.

Execute the remaining steps from the Administrator command prompt.

\(^{10}\)http://www.mongodb.org/downloads
Step 2: Install MongoDB for Windows.

Change to the directory containing the .msi installation binary of your choice and invoke:

```bash
msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.0.4-signed.msi
    INSTALLLOCATION="C:\mongodb"
    ADDLOCAL="all"
```

You can specify the installation location for the executable by modifying the \INSTALLLOCATION\ value.

By default, this method installs all MongoDB binaries. To install specific MongoDB component sets, you can specify them in the \ADDLOCAL\ argument using a comma-separated list including one or more of the following component sets:

<table>
<thead>
<tr>
<th>Component Set</th>
<th>Binaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>mongod.exe</td>
</tr>
<tr>
<td>Router</td>
<td>mongos.exe</td>
</tr>
<tr>
<td>Client</td>
<td>mongo.exe</td>
</tr>
<tr>
<td>MonitoringTools</td>
<td>mongostat.exe,mongotop.exe</td>
</tr>
<tr>
<td>ImportExportTools</td>
<td>mongodump.exe,mongorestore.exe,mongoexport.exe,</td>
</tr>
<tr>
<td></td>
<td>mongoinit.exe</td>
</tr>
<tr>
<td>MiscellaneousTools</td>
<td>bsondump.exe,mongofiles.exe,mongoplog.exe,mongoperf.exe</td>
</tr>
</tbody>
</table>

For instance, to install only the MongoDB utilities, invoke:

```bash
msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.0.4-signed.msi
    INSTALLLOCATION="C:\mongodb"
    ADDLOCAL="MonitoringTools,ImportExportTools,MiscellaneousTools"
```

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:

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

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

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

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

```bash
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 320).

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\(^{11}\) for more information.

Step 4: Begin using MongoDB.

To help you start using MongoDB, MongoDB provides *Getting Started Guides* in various driver editions. See *getting-started* for the available editions.

Before deploying MongoDB in a production environment, consider the *Production Notes* (page 197) document.

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.

Press the *Win* key, type `cmd.exe`, and press Ctrl + Shift + Enter to run the *Command Prompt* as Administrator.

Execute the remaining steps from the Administrator command prompt.

Step 2: Configure directories and files.

Create a configuration file and a directory path for MongoDB log output (`logpath`):

Create a specific directory for MongoDB log files:

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

In the *Command Prompt*, create a configuration file for the `logpath` option for MongoDB:

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

\(^{11}\)https://docs.mongodb.org/ecosystem/drivers/csharp
Step 3: Run the MongoDB service.

**Important:** Run all of the following commands in Command Prompt with “Administrative Privileges”

Install the MongoDB service. For --install to succeed, you must specify the logpath run-time option.

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

Modify the path to the mongod.cfg file as needed.

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 the dbpath directory does not exist, mongod.exe will not start. The default value for dbpath is \data\db.

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 4: 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\mongodb.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.**

Press the Win key, type cmd.exe, and press Ctrl + Shift + Enter to run the Command Prompt as Administrator.

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. This file can include any of the configuration options for mongod, but must include a valid setting for logpath:

The following creates a configuration file, specifying both the logpath and the dbpath settings in the configuration file:

```
echo logpath=c:\data\log\mongod.log > "C:\mongodb\mongod.cfg"
echo dbpath=c:\data\db >> "C:\mongodb\mongod.cfg"
```

Step 4: Create the MongoDB service.

Create the MongoDB service.

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

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[^12]
- MongoDB for .NET Developers Free Online Course[^13]
- MongoDB Architecture Guide[^14]

[^12]: https://university.mongodb.com/courses/M101P/about?jmp=docs
[^13]: https://university.mongodb.com/courses/M101N/about?jmp=docs
2.3.4 Install MongoDB Enterprise

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

**Install MongoDB Enterprise on Red Hat** *(page 33)* Install the MongoDB Enterprise build and required dependencies on Red Hat Enterprise or CentOS Systems using packages.

**Install MongoDB Enterprise on Ubuntu** *(page 37)* Install the MongoDB Enterprise build and required dependencies on Ubuntu Linux Systems using packages.

**Install MongoDB Enterprise on Debian** *(page 41)* Install the MongoDB Enterprise build and required dependencies on Debian Linux Systems using packages.

**Install MongoDB Enterprise on SUSE** *(page 44)* Install the MongoDB Enterprise build and required dependencies on SUSE Enterprise Linux.

**Install MongoDB Enterprise on Amazon AMI** *(page 47)* Install the MongoDB Enterprise build and required dependencies on Amazon Linux AMI.

**Install MongoDB Enterprise on Windows** *(page 49)* Install the MongoDB Enterprise build and required dependencies using the .msi installer.

## Install MongoDB Enterprise on Red Hat Enterprise or CentOS

### Overview

Use this tutorial to install MongoDB Enterprise\(^{15}\) on Red Hat Enterprise Linux or CentOS Linux versions 5, 6, and 7 from .rpm packages.

This installation guide only supports 64-bit systems. See *Platform Support* (page 809) for details.

### Packages

MongoDB provides officially supported Enterprise packages in their own repository. This repository contains 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`.

---

\(^{15}\)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 3.0.4, there are no control scripts for `mongos`. The `mongos` process is used only in `sharding` (page 667). 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-lib krb5-libs lm_sensors net-snmp-libs openssl popt rpm-libs tcp_wrappers zlib
```

**Version 6**

```
yum install cyrus-sasl-lib krb5-libs net-snmp-libs openssl
```

**Version 7**

```
yum install cyrus-sasl-lib krb5-libs lm_sensors-libs net-snmp-agent-libs net-snmp-libs openssl-libs rpm-libs tcp_wrappers-libs
```

The default `/etc/mongod.conf` configuration file supplied by the 3.0 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
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 924), 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. 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 2.6.1 release of MongoDB:

```
sudo yum install -y mongodb-enterprise-2.6.1 mongodb-enterprise-server-2.6.1 mongodb-enterprise-shell-2.6.1 mongodb-enterprise-mongos-2.6.1 mongodb-enterprise-tools-2.6.1
```

**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-mongos
```

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

**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:

---

16https://repo.mongodb.com/yum/redhat/

17http://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 prevent mongod from 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 help you start using MongoDB, MongoDB provides `Getting Started Guides` in various driver editions. See `getting-started` for the available editions.

Before deploying MongoDB in a production environment, consider the `Production Notes` (page 197) document.

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

**Overview**

Use this tutorial to install MongoDB Enterprise¹⁸ on LTS Ubuntu Linux systems from `.deb` packages.

---

MongoDB only provides packages for 64-bit long-term support Ubuntu releases. Currently, this means 12.04 LTS (Precise Pangolin) and 14.04 LTS (Trusty Tahr). While the packages may work with other Ubuntu releases, this is not a supported configuration.

Packages

MongoDB provides officially supported Enterprise packages in their own repository. This repository contains 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, mongoplog, 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 3.0.4, there are no control scripts for mongos. The mongos process is used only in sharding (page 667). You can use the mongod init script to derive your own mongos control script.

Considerations

MongoDB only provides packages for 64-bit long-term support Ubuntu releases. Currently, this means 12.04 LTS (Precise Pangolin) and 14.04 LTS (Trusty Tahr). While the packages may work with other Ubuntu releases, this is not a supported configuration.

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.** 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:\(^{19}\):

```bash
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:

```bash
echo "deb http://repo.mongodb.com/apt/ubuntu 
$(lsb_release -sc) /mongodb-enterprise/stable multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-enterprise.list
```

If you’d like to install MongoDB Enterprise packages from a particular release series (page 924), 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:

```bash
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:

```bash
sudo apt-get update
```

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

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

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

**Install a specific release of MongoDB Enterprise.** To install a specific release, you must specify each component package individually along with the version number, as in the following example:

```bash
sudo apt-get install -y mongodb-enterprise=3.0.4 mongodb-enterprise-server=3.0.4 mongodb-enterprise-shell=3.0.4 mongodb-enterprise-mongos=3.0.4 mongodb-enterprise-tools=3.0.4
```

If you only install `mongodb-enterprise=3.0.4` and do not include the component packages, the latest version of each MongoDB package will be installed regardless of what version you specified.

**Pin a specific version of MongoDB Enterprise.** 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:

```bash
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
```

Versions of the MongoDB packages before 2.6 use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

---

19[https://docs.mongodb.org/10gen-gpg-key.asc]
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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

Overview

Use this tutorial to install MongoDB Enterprise from .deb packages on Debian 7 “Wheezy”. This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

Packages

MongoDB provides officially supported Enterprise packages in their own repository. This repository contains 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`, `mongodump`, `mongoplog`, `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 3.0.4, there are no control scripts for `mongos`. The `mongos` process is used only in sharding (page 667). You can use the `mongod` init script to derive your own `mongos` control script.

20https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

2.3. Installation Guides 41
Considerations

This installation guide only supports 64-bit systems. See *Platform Support* (page 809) for details.

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[^21] 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 wheezy/mongodb-enterprise/stable main" | sudo tee /etc/apt/sources.list.d/mongodb-enterprise.list
```

If you’d like to install MongoDB Enterprise packages from a particular *release series* (page 924), 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 wheezy/mongodb-enterprise/2.6 main" | sudo tee /etc/apt/sources.list.d/mongodb-enterprise-2.6.list
```

Currently packages are only available for Debian 7 (Wheezy).

**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.** You can install either the latest stable version of MongoDB or a specific version of MongoDB.

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

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

**Install a specific release of MongoDB Enterprise.** To install a specific release, you must specify each component package individually along with the version number, as in the following example:

```
sudo apt-get install -y mongodb-enterprise=3.0.4 mongodb-enterprise-server=3.0.4 mongodb-enterprise-shell=3.0.4 mongodb-enterprise-mongos=3.0.4 mongodb-enterprise-tools=3.0.4
```

If you only install `mongodb-enterprise=3.0.4` and do not include the component packages, the latest version of each MongoDB package will be installed regardless of what version you specified.

[^21]: https://docs.mongodb.org/10gen-gpg-key.asc
Pin a specific version of MongoDB Enterprise. 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-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
```

Versions of the MongoDB packages before 2.6 use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

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 help you start using MongoDB, MongoDB provides *Getting Started Guides* in various driver editions. See `getting-started` for the available editions.

Before deploying MongoDB in a production environment, consider the *Production Notes* (page 197) document.

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:

```sh
sudo service mongod stop
```

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

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

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

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

Install MongoDB Enterprise on SUSE

**Overview**

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

This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

**Packages**

MongoDB provides officially supported Enterprise packages in their own repository. This repository contains 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.

---

22https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
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 3.0.4, there are no control scripts for `mongos`. The `mongos` process is used only in sharding (page 667). 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 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.

To manually install all dependencies, run the following command:

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

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.

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 latest stable release of MongoDB.

```
```

If you’d like to install MongoDB packages from a previous release series (page 924), 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, use the following command:

```
sudo zypper addrepo --no-gpgcheck https://repo.mongodb.com/zypper/suse/11/mongodb-enterprise/2.6/x86_64/mongodb
```

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 zypper install mongodb-enterprise-tools
```

2.3. Installation Guides
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-3.0.4 mongodb-enterprise-server-3.0.4 mongodb-enterprise-shell-3.0.4 mongodb-enterprise-mongos-3.0.4 mongodb-enterprise-tools-3.0.4

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-3.0.4 mongodb-enterprise-server-3.0.4 mongodb-enterprise-shell-3.0.4 mongodb-enterprise-mongos-3.0.4 mongodb-enterprise-tools-3.0.4

Previous versions of MongoDB packages use a different repository location. Refer to the version of the documentation appropriate for your MongoDB version.

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 help you start using MongoDB, MongoDB provides Getting Started Guides in various driver editions. See getting-started for the available editions.

Before deploying MongoDB in a production environment, consider the Production Notes (page 197) document.

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

Overview

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

This installation guide only supports 64-bit systems. See Platform Support (page 809) for details.

Prerequisites

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

```
yum install cyrus-sasl-lib krb5-libs lm_sensors-libs net-snmp-agent-libs net-snmp-libs openssl-libs rpm-libs tcp_wrappers-libs
```

---

23https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
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 https://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 https://downloads.mongodb.com/linux/mongodb-linux-x86_64-enterprise-amzn64-3.0.4.tgz
tar -zxvf mongodb-linux-x86_64-enterprise-amzn64-3.0.4.tgz
cp -R -n mongodb-linux-x86_64-enterprise-amzn64-3.0.4/ 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 /data/db 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 /data/db 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:

```
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 help you start using MongoDB, MongoDB provides *Getting Started Guides* in various driver editions. See *getting-started* for the available editions.

Before deploying MongoDB in a production environment, consider the *Production Notes* (page 197) document. Later, to stop MongoDB, press `Control+C` in the terminal where the `mongod` instance is running.

**Install MongoDB Enterprise on Windows**

New in version 2.6.

**Overview**

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

**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*.

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

```
wmic os get caption
wmic os get osarchitecture
```

**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. 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. Change to the directory containing the .msi installation binary of your choice and invoke:

```
msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.0.4-signed.msi ^
INSTALLOCATION=C:\mongodb ^
ADDLOCAL=all
```

You can specify the installation location for the executable by modifying the INSTALLOCATION value.

By default, this method installs all MongoDB binaries. To install specific MongoDB component sets, you can specify them in the ADDLOCAL argument using a comma-separated list including one or more of the following component sets:

<table>
<thead>
<tr>
<th>Component Set</th>
<th>Binaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>mongod.exe</td>
</tr>
<tr>
<td>Router</td>
<td>mongos.exe</td>
</tr>
<tr>
<td>Client</td>
<td>mongo.exe</td>
</tr>
<tr>
<td>MonitoringTools</td>
<td>mongostat.exe,mongotop.exe</td>
</tr>
<tr>
<td>ImportExportTools</td>
<td>mongodump.exe,mongorestore.exe,mongoexport.exe,</td>
</tr>
<tr>
<td></td>
<td>mongoinport.exe</td>
</tr>
<tr>
<td>MiscellaneousTools</td>
<td>bsondump.exe,mongofiles.exe,mongooplog.exe,mongoperf.exe</td>
</tr>
</tbody>
</table>

For instance, to install only the MongoDB utilities, invoke:

```
msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.0.4-signed.msi ^
INSTALLOCATION=C:\mongodb ^
ADDLOCAL=MonitoringTools,ImportExportTools,MiscellaneousTools
```

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\mongodb\data"

You may also specify the \texttt{dbpath} in a configuration file.

\textbf{Step 2: Start MongoDB.} To start MongoDB, run \texttt{mongod.exe}. For example, from the \textit{Command Prompt}:

C:\mongodb\bin\mongod.exe

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

Depending on the security level of your system, Windows may pop up a \textit{Security Alert} dialog box about blocking “some features” of \texttt{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 \textit{Security Documentation} (page 320).

\textbf{Step 3: Connect to MongoDB.} To connect to MongoDB through the \texttt{mongo.exe} shell, open another \textit{Command Prompt}.

C:\mongodb\bin\mongo.exe

If you want to develop applications using .NET, see the documentation of C\ and MongoDB\footnote{https://docs.mongodb.org/ecosystem/drivers/csharp} for more information.

\textbf{Step 4: Begin using MongoDB.} To help you start using MongoDB, MongoDB provides \textit{Getting Started Guides} in various driver editions. See \textit{getting-started} for the available editions.

Before deploying MongoDB in a production environment, consider the \textit{Production Notes} (page 197) document.

Later, to stop MongoDB, press Control+C in the terminal where the \texttt{mongod} instance is running.

\textbf{Configure a Windows Service for MongoDB Enterprise}

\textbf{Step 1: Open an Administrator command prompt.} Press the \texttt{Win} key, type \texttt{cmd.exe}, and press \texttt{Ctrl + Shift + Enter} to run the \textit{Command Prompt} as Administrator.

Execute the remaining steps from the Administrator command prompt.

\textbf{Step 2: Configure directories and files.} Create a configuration file and a directory path for MongoDB log output (\texttt{logpath}): Create a specific directory for MongoDB log files:

\texttt{md "C:\mongodb\log"}

In the \textit{Command Prompt}, create a configuration file for the \texttt{logpath} option for MongoDB:

\texttt{echo logpath=C:\mongodb\log\mongo.log > "C:\mongodb\mongodb.cfg"}
Step 3: Run the MongoDB service.

Important: Run all of the following commands in Command Prompt with “Administrative Privileges”

Install the MongoDB service. For --install to succeed, you must specify the logpath run-time option.

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

Modify the path to the mongod.cfg file as needed.

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

If the dbpath directory does not exist, mongod.exe will not start. The default value for dbpath is `\data\db`.

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 4: 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.** Press the Win key, type `cmd.exe`, and press Ctrl + Shift + Enter to run the Command Prompt as Administrator.

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. This file can include any of the configuration options for mongod, but must include a valid setting for logpath:

The following creates a configuration file, specifying both the logpath and the dbpath settings in the configuration file:

```
echo logpath=c:\data\log\mongod.log> "C:\mongodb\mongod.cfg"
echo dbpath=c:\data\db>> "C:\mongodb\mongod.cfg"
```
Step 4: Create the MongoDB service. Create the MongoDB service.

```
sc.exe create MongoDB binPath= "C:\mongodb\bin\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.

```
network start MongoDB
```

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

```
network stop MongoDB
```

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

```
sc.exe delete MongoDB
```

2.3.5 Verify Integrity of MongoDB Packages

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 should validate the package using either the provided PGP signature or SHA-256 checksum.

PGP signatures provide the strongest guarantees by checking both the authenticity and integrity of a file to prevent tampering.

Cryptographic checksums only validate file integrity to prevent network transmission errors.

Procedures

Use PGP/GPG

MongoDB signs each release branch with a different PGP key. The public key files for each release branch since MongoDB 2.2 are available for download from the key server in both textual .asc and binary .pub formats.

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
```

27https://www.mongodb.org/static/pgp/
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-256

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 3.0.4 release for OS X through the shell, type this command:

```
curl -LO https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.4.tgz
```

Step 2: Download the SHA256 file.

```
curl -LO https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.4.tgz.sha256
```

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

```
shasum -c mongodb-osx-x86_64-3.0.4.tgz.sha256
```

which should return the following if the checksum matched the downloaded package:
2.4 First Steps with MongoDB

Once you have installed MongoDB, consider the Getting Started Guides to learn about MongoDB. The guides are available in various driver editions.

2.5 Additional Resources

- Install MongoDB using MongoDB Cloud Manager
- Create a New MongoDB Deployment with Ops Manager: Ops Manager is an on-premise solution available in MongoDB Enterprise Advanced.
- MongoDB CRUD Concepts (page 60)
- Data Models (page 143)
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 57)** An introduction to the MongoDB data model as well as queries and data manipulations.

**MongoDB CRUD Concepts (page 60)** The core documentation of query and data manipulation.

**MongoDB CRUD Tutorials (page 92)** Examples of basic query and data modification operations.

**MongoDB CRUD Reference (page 129)** Reference material for the query and data manipulation interfaces.

### 3.1 MongoDB CRUD Introduction

MongoDB stores data in the form of documents, which are JSON-like field and value pairs. Documents are analogous to structures in programming languages that associate keys with values (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 168).

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

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:

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1} )
```

See *Read Operations Overview* (page 61) 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 74) 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
an ordered representation of all values of a field or a group of fields. Indexes may also enforce uniqueness (page 504),
store objects in a geospatial representation (page 494), and facilitate text search (page 500).
Replica Set Read Preference

For replica sets and sharded clusters with replica set components, applications specify read preferences (page 588). 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 78). 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 60) and Write Operations (page 73) documents introduce the behavior and operations of read and write operations for MongoDB deployments.

Read Operations (page 60) Queries are the core operations that return data in MongoDB. Introduces queries, their behavior, and performances.

Cursors (page 64) Queries return iterable objects, called cursors, that hold the full result set.

Query Optimization (page 65) Analyze and improve query performance.

Distributed Queries (page 69) Describes how sharded clusters and replica sets affect the performance of read operations.

Write Operations (page 73) Write operations insert, update, or remove documents in MongoDB. Introduces data create and modify operations, their behavior, and performances.

Write Concern (page 78) Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

Distributed Write Operations (page 82) 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 73) 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 61) A high level overview of queries and projections in MongoDB, including a discussion of syntax and behavior.

Cursors (page 64) Queries return iterable objects, called cursors, that hold the full result set.

Query Optimization (page 65) Analyze and improve query performance.

Query Plans (page 68) MongoDB executes queries using optimal plans.
Read Operations Overview

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 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:

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

collection
query criteria
projection
cursor modifier

The next diagram shows the same query in SQL:

```
SELECT _id, name, address  
FROM users  
WHERE age > 18  
LIMIT 5
```

projection
table
select criteria
cursor modifier

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 62) for details.

See

*SQL to MongoDB Mapping Chart* (page 131) 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 102) (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 96).

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

```javascript
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 107) 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**

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 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:

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

To manually iterate the cursor to access the documents, see [Iterate a Cursor in the mongo Shell](page 110).

**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:

```javascript
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 754).

---

1 You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value 20. See [Executing Queries](page 273) for more information.
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:

```javascript
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:

```javascript
db.serverStatus().metrics.cursor
```

The result is the following document:

```javascript
{
   "timedOut" : <number>,
   "open" : {
      "noTimeout" : <number>,
      "pinned" : <number>,
      "total" : <number>
   }
}
```

See also:

`db.serverStatus()`

Query Optimization

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

If your application queries a collection on a particular field or 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 486).

Example

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

```javascript
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. In the mongo shell, you can create indexes using the `db.collection.createIndex()` method:

```javascript
db.inventory.createIndex( { 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 111).

In addition to optimizing read operations, indexes can support sort operations and allow for a more efficient storage utilization. See `db.collection.createIndex()` and Indexing Tutorials (page 517) for more information about index creation.

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 66) a query when both of the following apply:

- all the fields in the query (page 96) 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:

---

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.
db.inventory.createIndex( { 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 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**

**Restrictions on Indexed Fields**  An index **cannot** cover a query if:

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* (page 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.

**Restrictions on Sharded Collection**  An index cannot cover a query on a *sharded* collection when run against a mongos if the index does not contain the shard key, with the following exception for the _id index: If a query on a sharded collection only specifies a condition on the _id field and returns only the _id field, the _id index can cover the query when run against a mongos even if the _id field is not the shard key.

Changed in version 3.0: In previous versions, an index cannot cover (page 66) a query on a sharded collection when run against a mongos.

**explain**  To determine whether a query is a covered query, use the `db.collection.explain()` or the `explain()` method and review the results.

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

³ To index fields in embedded documents, use *dot notation*. 

3.2. MongoDB CRUD Concepts 67
For more information see *Measure Index Use* (page 528).

**Query Plans**

The MongoDB query optimizer processes queries and chooses the most efficient query plan for a query given the available indexes. The query system then uses this query plan each time the query runs.

The query optimizer 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 69).

You can use the `db.collection.explain()` or the `cursor.explain()` method to view statistics about the query plan for a given query. This information can help as you develop *indexing strategies* (page 546).

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

**Query Optimization**

To create a new query plan, the query optimizer:

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

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

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

```javascript
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.


**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 shape, check the `indexFilterSet` field of either the `db.collection.explain()` or the `cursor.explain()` method.

Index filters only affect 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**

**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 661) 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 674). When a query includes a shard key, the `mongos` can use cluster metadata from the *config database* (page 670) 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 678) and *Shard Keys* (page 674) 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 657).

You can configure the read preference mode (page 657) on a per-connection or per-operation basis to allow reads from secondaries to:

- reduce latency in multi-data-center deployments,
- improve read throughput by distributing high read-volumes (relative to write volume),
- for backup operations, and/or
- to allow reads during failover (page 580) 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. 

For more information on read preference or on the read preference modes, see Read Preference (page 588) and Read Preference Modes (page 657).

3.2.2 Write Operations

The following documents describe 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 \texttt{w: majority} write concern (page 130). The node that can complete \texttt{w: majority} (page 130) 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 657).
Write Operations Overview (page 74) Provides an overview of MongoDB’s data insertion and modification operations, including aspects of the syntax, and behavior.

Write Concern (page 78) Describes the kind of guarantee MongoDB provides when reporting on the success of a write operation.

Atomicity and Transactions (page 80) Describes write operation atomicity in MongoDB.

Distributed Write Operations (page 82) Describes how MongoDB directs write operations on sharded clusters and replica sets and the performance characteristics of these operations.

Write Operation Performance (page 86) Introduces the performance constraints and factors for writing data to MongoDB deployments.

Bulk Write Operations (page 87) Provides an overview of MongoDB’s bulk write operations.

Storage (page 89) Introduces the storage allocation strategies available for MongoDB collections.

Write Operations Overview

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 74), update (page 75), and remove (page 77). 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 60).

MongoDB allows applications to determine the acceptable level of acknowledgement required of write operations. See Write Concern (page 78) 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:

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

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.

```javascript
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 78), 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 76) 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 76).

**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:

**Example**

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

3.2. MongoDB CRUD Concepts 75
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 `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:

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

The following diagram shows the same query in SQL:

```
DELETE FROM users           ← table
WHERE status = 'D'          ← delete criteria
```

### Example

```javascript
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 106).

**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 80).

**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

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 852) integrates write concern with the write operations.

For details on write concern configurations, see Write Concern Reference (page 130).

Considerations

Default Write Concern  The <code>mongo</code> shell and the MongoDB drivers use Acknowledged (page 78) as the default write concern.

See Acknowledged (page 78) for more information, including when this write concern became the default.

Timeouts   Clients can set a <code>wtimeout</code> (page 131) value as part of a replica acknowledged (page 80) 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 <code>wtimeout</code> 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 924), 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 924).

Changed in version 2.6: The <code>mongo</code> shell write methods now incorporates the write concern (page 78) in the write methods and provide the default write concern whether run interactively or in a script. See Write Method Acknowledgements (page 857) 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 `storage.mmapv1.journal.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 586) 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 130)

**Atomicity and Transactions**

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 isolate-update-operations. 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 115) 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 115).

### 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 504) 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 122).

The two-phase commit pattern provides a variation where the query predicate includes the application identifier (page 120) as well as the expected state of the data in the write operation.

### Distributed Write Operations

#### Write Operations on Sharded Clusters

For sharded collections in a sharded cluster, the mongos directs write operations from applications to the shards that are responsible for the specific portion of the data set. The mongos uses the cluster metadata from the config database (page 670) 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.
3.2. MongoDB CRUD Concepts
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 690) and Bulk Write Operations (page 87).

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 580) in the form of rollbacks (page 584) as well as general read consistency (page 585).

To help avoid this issue, you can customize the write concern (page 78) to return confirmation of the write operation to another member 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 80), Oplog Size (page 593), and Change the Size of the Oplog (page 628).

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 596) deployments, MongoDB treats w: "majority" as equivalent to w: 1. In earlier versions of MongoDB, w: "majority" produces an error in master/slave (page 596) deployments.
Write Operation Performance

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.  

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 65). For more information on indexes, see Indexes (page 481) and Indexing Strategies (page 546).

Document Growth and the MMAPv1 Storage Engine

Some update operations can increase the size of the document; for instance, if an update adds a new field to the document. 

For the MMAPv1 storage engine, 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. Updates that require relocations take longer than updates that do not, 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. 

Changed in version 3.0.0: By default, MongoDB uses Power of 2 Sized Allocations (page 91) to add padding automatically (page 91) for the MMAPv1 storage engine. The Power of 2 Sized Allocations (page 91) 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 as well as reduce the occurrences of reallocations in many cases. 

Although Power of 2 Sized Allocations (page 91) minimizes the occurrence of re-allocation, it does not eliminate document re-allocation. 

See Storage (page 89) 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 197) for recommendations regarding additional hardware and configuration options.

Journaling

MongoDB uses write ahead logging to an on-disk journal to guarantee write operation durability and to provide crash resiliency. Before applying a change to the data files, MongoDB writes the change operation to the journal. 

For inserts and updates to un-indexed fields, the overhead for sparse indexes (page 505) is less than for non-sparse indexes. Also for non-sparse indexes, updates that do not change the record size have less indexing overhead.

---

6 For inserts and updates to un-indexed fields, the overhead for sparse indexes (page 505) 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 78) that includes journaled (page 80), 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 308).

## Bulk Write Operations

### 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()` method. To see the documentation for earlier versions, see Bulk Inserts.

## 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.

### 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:

   7http://docs.mongodb.org/v2.4/core/bulk-inserts
   8http://docs.mongodb.org/v2.4/core/bulk-inserts
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-bulk` 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 852)

**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 723).

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 723).

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.

```c++
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 674) for information on choosing a sharded key. Also see Shard Key Internals (page 674) (in particular, Choosing a Shard Key (page 694)).

Storage
New in version 3.0: MongoDB adds support for additional storage engines. MongoDB’s original storage engine, known as mmapv1 remains the default in 3.0, but the new wiredTiger engine is available and can offer additional flexibility and improved throughput for many workloads.

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.

All documents are part of a collection, which are a logical groupings 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. A single MongoDB deployment may have many databases.

### WiredTiger Storage Engine

New in version 3.0.

WiredTiger is a storage engine that is optionally available in the 64-bit build of MongoDB 3.0. It excels at read and insert workloads as well as more complex update workloads.

#### Document Level Locking

With WiredTiger, all write operations happen within the context of a document level lock. As a result, multiple clients can modify more than one document in a single collection at the same time. With this very granular concurrency control, MongoDB can more effectively support workloads with read, write and updates as well as high-throughput concurrent workloads.

#### Journal

WiredTiger uses a write-ahead transaction log in combination with checkpoints to ensure data persistence. With WiredTiger, MongoDB will commit a checkpoint to disk every 60 seconds or when there are 2 gigabytes of data to write. Between and during checkpoints the data files are always valid.

The WiredTiger journal persists all data modifications between checkpoints. If MongoDB exits between checkpoints, it uses the journal to replay all data modified since the last checkpoint. By default, the WiredTiger journal is compressed using the snappy compression library. To specify an alternate compression algorithm or no compression, use the storage.wiredTiger.engineConfig.journalCompressor setting.

You can disable journaling by setting storage.journal.enabled to false, which can reduce the overhead of maintaining the journal. For standalone instances, not using the journal means that you will lose some data modifications when MongoDB exits unexpectedly between checkpoints. For members of replica sets, the replication process may provide sufficient durability guarantees.

#### Compression

With WiredTiger, MongoDB supports compression for all collections and indexes. Compression minimizes storage use at the expense of additional CPU.

By default, WiredTiger uses block compression with the snappy compression library for all collections and prefix compression for all indexes.

For collections, block compression with zlib is also available. To specify an alternate compression algorithm or no compression, use the storage.wiredTiger.collectionConfig.blockCompressor setting.

For indexes, to disable prefix compression, use the storage.wiredTiger.indexConfig.prefixCompression setting.

Compression settings are also configurable on a per-collection and per-index basis during collection and index creation. See create-collection-storage-engine-options and db.collection.createIndex() storageEngine option.

For most workloads, the default compression settings balance storage efficiency and processing requirements.

The WiredTiger journal is also compressed by default. For information on journal compression, see Journal (page 90).

See also:

http://wiredtiger.com
MMAPv1 Storage Engine

MMAPv1 is MongoDB’s original storage engine based on memory mapped files. It excels at workloads with high volume inserts, reads, and in-place updates. MMAPv1 is the default storage engine in MongoDB 3.0 and all previous versions.

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 308) for more information about the journal in MongoDB.

Record Storage Characteristics 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.

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 91) are more efficient for insert/update/delete workloads; while exact fit allocations (page 91) is ideal for collections without update and delete workloads.

Power of 2 Sized Allocations Changed in version 3.0.0.

MongoDB 3.0 uses the power of 2 sizes allocation as the default record allocation strategy for MMAPv1. With the power of 2 sizes allocation strategy, each record has a size in bytes that is a power of 2 (e.g. 32, 64, 128, 256, 512 ... 2MB). For documents larger than 2MB, the allocation is rounded up to the nearest multiple of 2MB.

The power of 2 sizes allocation strategy has the following key properties:

- Can efficiently reuse freed records to reduce fragmentation. Quantizing record allocation sizes into a fixed set of sizes increases the probability that an insert will fit into the free space created by an earlier document deletion or relocation.

- Can reduce moves. The added padding space gives a document room to grow without requiring a move. In addition to saving the cost of moving, this results in less updates to indexes. Although the power of 2 sizes strategy can minimize moves, it does not eliminate them entirely.

No Padding Allocation Strategy Changed in version 3.0.0.

For collections whose workloads do not change the document sizes, such as workloads that consist of insert-only operations or update operations that do not increase document size (such as incrementing a counter), you
can disable the power of 2 allocation (page 91) using the `collMod` command with the `noPadding` flag or the `db.createCollection()` method with the `noPadding` option.

Prior to version 3.0.0, MongoDB used an allocation strategy that included a dynamically calculated padding as a factor of the document size.

**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 209) 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 57).

**Insert Documents** (page 92) Insert new documents into a collection.

**Query Documents** (page 96) Find documents in a collection using search criteria.

**Modify Documents** (page 102) Modify documents in a collection

**Remove Documents** (page 106) Remove documents from a collection.

**Limit Fields to Return from a Query** (page 107) Limit which fields are returned by a query.

**Limit Number of Elements in an Array after an Update** (page 109) Use `$push` with modifiers to sort and maintain an array of fixed size.

**Iterate a Cursor in the _mongo_ Shell** (page 110) Access documents returned by a `find` query by iterating the cursor, either manually or using the iterator index.

**Analyze Query Performance** (page 111) 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 115) Use two-phase commits when writing data to multiple documents.

**Update Document if Current** (page 122) Update a document only if it has not changed since it was last read.

**Create Tailable Cursor** (page 123) 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 125) 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

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.

```javascript
db.inventory.insert(
  {
    item: "ABC1",
    details: {
      model: "14Q3",
      manufacturer: "XYZ Company"
    },
    category: "clothing"
  }
)
```

The operation returns a `WriteResult` object with the status of the operation. A successful insert of the document returns the following object:

```javascript
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.

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

The document you inserted should return.

```javascript
{ "_id" : ObjectId("53d98f133bb604791249ca99"), "item" : "ABC1", "details" : { "model" : "14Q3", "manufacturer" : "XYZ Company" }, "stock" : [ { "size" : "S", "qty" : 25 }, { "size" : "M", "qty" : 50 } ], "category" : "clothing" }
```

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` 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.

```javascript
var mydocuments = [
  {

default references:
\(^9\)https://docs.mongodb.org/manual/reference/object-id
\(^10\)https://docs.mongodb.org/manual/reference/object-id
Step 2: Insert the documents.

Pass the `mydocuments` array to the `db.collection.insert()` to perform a bulk insert.

```javascript
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:

```json
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.

```javascript
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" },
    category: "houseware"
  });
```

**Step 3: Execute the bulk operation.**

Call the `execute()` method on the bulk object to execute the operations in its list.

```javascript
bulk.execute();
```

The method returns a `BulkWriteResult` object with the status of the operation. A successful insert of the documents returns the following object:

```javascript
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

In MongoDB, the `db.collection.find()` method retrieves documents from a collection. The `db.collection.find()` method returns a cursor (page 64) to the retrieved documents.

This tutorial provides examples of read operations using the `db.collection.find()` method in the `mongo` shell. In these examples, the retrieved documents contain all their fields. To restrict the fields to return in the retrieved documents, see Limit Fields to Return from a Query (page 107).

Select All Documents in a Collection

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

```javascript
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:

```javascript
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`:

```javascript
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':

```javascript
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.

---

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.
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**:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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 99).

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 99).

Consider an inventory collection that contains the following documents:

```json
{ _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:

```javascript
db.inventory.find( { ratings: [ 5, 8, 9 ] } )
```
The operation returns the following document:

```json
{ "_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:

```json
{ "_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:

```json
{ "_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:

```json
{ "_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`:

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

The operation returns the following documents:

```json
{
    _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`:

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

The operation returns the following document:

```json
{
    _id: 100,
    type: "food",
    item: "xyz",
    qty: 25,
```
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:

```javascript
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 107)

### 3.3.3 Modify Documents

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\(^\text{12}\), such as \$set to modify values.

Some update operators, such as \$set, will create the field if the field does not exist. See the individual update operator\(^\text{13}\) 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.

```javascript
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.

```javascript
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.

\(^\text{12}\)https://docs.mongodb.org/manual/reference/operator/update

\(^\text{13}\)https://docs.mongodb.org/manual/reference/operator/update
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".

```
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.

```javascript
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.

```javascript
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.

```javascript
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

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`:

```javascript
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

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:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;field&gt;</code>: <code>&lt;1 or true&gt;</code></td>
<td>Specify the inclusion of a field.</td>
</tr>
<tr>
<td><code>&lt;field&gt;</code>: <code>&lt;0 or false&gt;</code></td>
<td>Specify the suppression of the field.</td>
</tr>
</tbody>
</table>

**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 64) to the retrieved documents. For examples on query selection criteria, see `Query Documents` (page 96).

Return All Fields in Matching Documents

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

```javascript
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.

```javascript
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:

```javascript
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 171) 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.
3.3.6 Limit Number of Elements in an Array after an Update

New in version 2.4.

Synopsis

Consider an application where users may submit many scores (e.g. for a test), but the application only needs to track the top three test scores.

This pattern uses the \$push operator with the \$each, \$sort, and \$slice modifiers to sort and maintain an array of fixed size.

Pattern

Consider the following document in the collection students:

```json
{  
  _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.

```javascript
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

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:

```javascript
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:

```javascript
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())`:

```javascript
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:

---

14 You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value 20. See [Executing Queries](page 273) for more information.
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:

```javascript
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:

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

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

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

### 3.3.8 Analyze Query Performance

The `cursor.explain("executionStats")` and the `db.collection.explain("executionStats")` methods provide statistics about the performance of a query. This data output can be useful in measuring if and how a query uses an index.

`db.collection.explain()` provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

**Evaluate the Performance of a Query**

Consider a collection `inventory` with the following documents:

```json
{ "_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:

```javascript
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("executionStats")` method:

```javascript
db.inventory.find( { quantity: { $gte: 100, $lte: 200 } } ).explain("executionStats")
```

`explain()` returns the following results:

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    ...
    "winningPlan" : {
      "stage" : "COLLSCAN",
      ...
    }
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 3,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 10,
    "executionStages" : {
      "stage" : "COLLSCAN",
      ...
    },
    ...
  }
}
```

- `queryPlanner.winningPlan.stage` displays `COLLSCAN` to indicate a collection scan.
- `executionStats.nReturned` displays 3 to indicate that the query matches and returns three documents.
- `executionStats.totalDocsExamined` 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 of examined documents 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:
To view the query plan statistics, use the `explain("executionStats")` method:

```javascript
db.inventory.find(
    { quantity: { $gte: 100, $lte: 200 } }
).explain("executionStats")
```

The `explain()` method returns the following results:

```javascript
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        ...  
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "quantity" : 1
                },
                ...
            },
            ...
        },
        "rejectedPlans" : []
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 3,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 3,
        "totalDocsExamined" : 3,
        "executionStages" : {
            ...
        },
        ...
    }
}
```

- `queryPlanner.winningPlan.inputStage.stage` displays `IXSCAN` to indicate index use.
- `executionStats.nReturned` displays 3 to indicate that the query matches and returns three documents.
- `executionStats.totalKeysExamined` display 3 to indicate that MongoDB scanned three index entries.
- `executionStats.totalDocsExamined` display 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.createIndex( { quantity: 1, type: 1 } )
db.inventory.createIndex( { type: 1, quantity: 1 } )

Evaluate the effect of the first index on the query:

```javascript
db.inventory.find(
    { quantity: { $gte: 100, $lte: 300 }, type: "food" }
).hint({ quantity: 1, type: 1 }).explain("executionStats")
```

The explain() method returns the following output:

```json
{
    "queryPlanner" : {
        "..."
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "quantity" : 1,
                    "type" : 1
                },
                ...
            }
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 2,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 5,
        "totalDocsExamined" : 2,
        "executionStages" : {
            ...
        }
    }
}
```

MongoDB scanned 5 index keys (executionStats.totalKeysExamined) to return 2 matching documents (executionStats.nReturned).

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 }).explain("executionStats")

The `explain()` method returns the following output:

```json
{
    "queryPlanner" : {
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "type" : 1,
                    "quantity" : 1
                },
                ...,
            }
            ...,
        },
        "rejectedPlans" : [
        ],
        "executionStats" : {
            "executionSuccess" : true,
            "nReturned" : 2,
            "executionTimeMillis" : 0,
            "totalKeysExamined" : 2,
            "totalDocsExamined" : 2,
            "executionStages" : {
                ...,
            }
        },
        ...
    }
}
```

MongoDB scanned 2 index keys (executionStats.totalKeysExamined) to return 2 matching documents (executionStats.nReturned).

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 65), `Query Plans` (page 68), `Optimize Query Performance` (page 216), `Indexing Strategies` (page 546)

### 3.3.9 Perform Two Phase Commits

**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 120) 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 120). 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.

```javascript
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:

```javascript
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 117) step. If the collection contains additional documents, the query will return any transaction with an `initial` state unless you specify additional query conditions.

```javascript
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:

```javascript
```

**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.

```javascript
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 \( \_id \).

```javascript
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 \( \_id \).

```javascript
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:

```javascript
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.

```javascript
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.

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
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 117) 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.

```javascript
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.

```javascript
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.

```javascript
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.

```javascript
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:

```javascript
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 119), 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:

```javascript
var dateThreshold = new Date();
dateThreshold.setMinutes(dateThreshold.getMinutes() - 30);
```

```javascript
db.transactions.find{
```
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 78) for your deployment.

3.3.10 Update Document if Current

Overview

The Update if Current pattern is an approach to concurrency control (page 82) 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.

```javascript
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 125).

For more approaches, see Concurrency Control (page 82).

3.3.11 Create Tailable Cursor

Overview

By default, MongoDB will automatically close a cursor when the client has exhausted all results in the cursor. However, for capped collections (page 209) 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 559) 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.

```cpp
#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:

```cpp
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:

```cpp
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\(^{15}\)

### 3.3.12 Create an Auto-Incrementing Sequence Field

**Synopsis**

MongoDB reserves the `_id` field in the top level of all documents as a primary key. `_id` must be unique, and always has an index with a unique constraint (page 504). 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 126)
- **Optimistic Loop** (page 127)

\(^{15}\)http://shtylman.com/post/the-tail-of-mongodb
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.

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:

   ```javascript
   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:

   ```javascript
   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().

   ```javascript
   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():

   ```javascript
   db.users.find()
   ```

The _id fields contain incrementing sequence values:
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:

```javascript
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.

   ```javascript
   function insertDocument(doc, targetCollection) {
     while (1) {
       // Insert document with calculated _id value
       // Check status
     }
   }
   ```
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:

   ```javascript
   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:
The `while` loop may iterate many times in collections with larger insert volumes.

### 3.4 MongoDB CRUD Reference

#### 3.4.1 Query Cursor Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cursor.count()</td>
<td>Returns the total number of documents in a cursor.</td>
</tr>
<tr>
<td>cursor.explain()</td>
<td>Reports on the query execution plan, including index use, for a cursor.</td>
</tr>
<tr>
<td>cursor.hint()</td>
<td>Forces MongoDB to use a specific index for a query.</td>
</tr>
<tr>
<td>cursor.limit()</td>
<td>Constrains the size of a cursor’s result set.</td>
</tr>
<tr>
<td>cursor.next()</td>
<td>Returns the next document in a cursor.</td>
</tr>
<tr>
<td>cursor.skip()</td>
<td>Returns a cursor that begins returning results only after passing or skipping a number of documents.</td>
</tr>
<tr>
<td>cursor.sort()</td>
<td>Returns results ordered according to a sort specification.</td>
</tr>
<tr>
<td>cursor.toArray()</td>
<td>Returns an array that contains all documents returned by the cursor.</td>
</tr>
</tbody>
</table>

#### 3.4.2 Query and Data Manipulation Collection Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.collection.count()</td>
<td>Wraps <code>count</code> to return a count of the number of documents in a collection or matching a query.</td>
</tr>
<tr>
<td>db.collection.distinct()</td>
<td>Returns an array of documents that have distinct values for the specified field.</td>
</tr>
<tr>
<td>db.collection.find()</td>
<td>Performs a query on a collection and returns a cursor object.</td>
</tr>
<tr>
<td>db.collection.findOne()</td>
<td>Performs a query and returns a single document.</td>
</tr>
<tr>
<td>db.collection.insert()</td>
<td>Creates a new document in a collection.</td>
</tr>
<tr>
<td>db.collection.remove()</td>
<td>Deletes documents from a collection.</td>
</tr>
<tr>
<td>db.collection.save()</td>
<td>Provides a wrapper around an <code>insert()</code> and <code>update()</code> to insert new documents.</td>
</tr>
<tr>
<td>db.collection.update()</td>
<td>Modifies a document in a collection.</td>
</tr>
</tbody>
</table>

#### 3.4.3 MongoDB CRUD Reference Documentation

*Write Concern Reference* (page 130) Configuration options associated with the guarantee MongoDB provides when reporting on the success of a write operation.

*SQL to MongoDB Mapping Chart* (page 131) An overview of common database operations showing both the MongoDB operations and SQL statements.

*The bios Example Collection* (page 137) Sample data for experimenting with MongoDB. `insert()`, `update()` and `find()` pages use the data for some of their examples.
Write Concern Reference

Write concern (page 78) 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 852) 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 130) option to specify the required number of acknowledgments before returning, the `j` (page 130) option to require writes to the journal before returning, and `wtimeout` (page 131) 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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Provides acknowledgment of write operations on a standalone <code>mongod</code> or the primary in a replica set. This is the default write concern for MongoDB.</td>
</tr>
<tr>
<td>0</td>
<td>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 <code>mongod</code> acknowledge the write operation.</td>
</tr>
<tr>
<td>&lt;Number greater than 1&gt;</td>
<td>Guarantees that write operations have propagated successfully to the specified number of replica set members including the primary. For example, <code>w: 2</code> indicates acknowledgements from the primary and at least one secondary. If you set <code>w</code> 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.</td>
</tr>
<tr>
<td>&quot;majority&quot;</td>
<td>Confirms that write operations have propagated to the majority of voting nodes: a majority of the replica set’s voting 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 3.0: In previous versions, <code>w: &quot;majority&quot;</code> refers to the majority of the replica set’s members. Changed in version 2.6: In <code>Master/Slave</code> (page 596) deployments, MongoDB treats <code>w: &quot;majority&quot;</code> as equivalent to <code>w: 1</code>. In earlier versions of MongoDB, <code>w: &quot;majority&quot;</code> produces an error in <code>master/slave</code> (page 596) deployments. By specifying a <code>tag set</code> (page 635), you can have fine-grained control over which replica set members must acknowledge a write operation to satisfy the required level of write concern.</td>
</tr>
<tr>
<td>&lt;tag set&gt;</td>
<td></td>
</tr>
</tbody>
</table>

**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 78) and *Write Concern for Replica Sets* (page 80).

**SQL to MongoDB Mapping Chart**

In addition to the charts that follow, you might want to consider the *Frequently Asked Questions* (page 743) 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.

<table>
<thead>
<tr>
<th>SQL Terms/Concepts</th>
<th>MongoDB Terms/Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row</td>
<td>document or BSON document</td>
</tr>
<tr>
<td>column</td>
<td>field</td>
</tr>
<tr>
<td>index</td>
<td>index</td>
</tr>
<tr>
<td>table joins</td>
<td>embedded documents and linking</td>
</tr>
<tr>
<td>primary key</td>
<td>primary key</td>
</tr>
<tr>
<td>Specify any unique column or column combination as primary key.</td>
<td>In MongoDB, the primary key is automatically set to the <code>_id</code> field.</td>
</tr>
<tr>
<td>aggregation (e.g. group by)</td>
<td>aggregation pipeline</td>
</tr>
<tr>
<td>See the <em>SQL to Aggregation Mapping Chart</em> (page 476).</td>
<td></td>
</tr>
</tbody>
</table>

**Executables**

The following table presents some database executables and the corresponding MongoDB executables. This table is not meant to be exhaustive.

<table>
<thead>
<tr>
<th>Executables</th>
<th>MongoDB</th>
<th>MySQL</th>
<th>Oracle</th>
<th>Informix</th>
<th>DB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Server</td>
<td>mongod</td>
<td>mysql</td>
<td>oracle</td>
<td>IDS</td>
<td>DB2 Server</td>
</tr>
<tr>
<td>Database Client</td>
<td>mongo</td>
<td>mysql</td>
<td>sqlplus</td>
<td>DB-Access</td>
<td>DB2 Client</td>
</tr>
</tbody>
</table>

---

3.4. MongoDB CRUD Reference 131
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:

```json
{
    _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.
<table>
<thead>
<tr>
<th>SQL Schema Statements</th>
<th>MongoDB Schema Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE users (</td>
<td>Implicitly created on first insert() operation. The primary key _id is automatically added if _id field is not specified.</td>
</tr>
<tr>
<td>id MEDIUMINT NOT NULL</td>
<td>db.users.insert( {</td>
</tr>
<tr>
<td>AUTO_INCREMENT,</td>
<td>user_id: &quot;abc123&quot;,</td>
</tr>
<tr>
<td>user_id Varchar(30),</td>
<td>age: 55,</td>
</tr>
<tr>
<td>age Number,</td>
<td>status: &quot;A&quot;</td>
</tr>
<tr>
<td>status char(1),</td>
<td>} )</td>
</tr>
<tr>
<td>PRIMARY KEY (id)</td>
<td>However, you can also explicitly create a collection:</td>
</tr>
<tr>
<td>)</td>
<td>db.createCollection(&quot;users&quot;)</td>
</tr>
<tr>
<td></td>
<td>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</td>
</tr>
<tr>
<td>ALTER TABLE users</td>
<td>However, at the document level, update() operations can add fields to existing documents using the $set operator.</td>
</tr>
<tr>
<td>ADD join_date DATETIME</td>
<td>db.users.update( {</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>{ $set: { join_date: new Date() } },</td>
</tr>
<tr>
<td></td>
<td>{ multi: true }</td>
</tr>
<tr>
<td></td>
<td>} )</td>
</tr>
<tr>
<td>ALTER TABLE users</td>
<td>Collections do not describe or enforce the structure of its documents; i.e. there is no structural alteration at the collection level.</td>
</tr>
<tr>
<td>DROP COLUMN join_date</td>
<td>However, at the document level, update() operations can remove fields from documents using the $unset operator.</td>
</tr>
<tr>
<td></td>
<td>db.users.update( {</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>{ $unset: { join_date: &quot;&quot; } },</td>
</tr>
<tr>
<td></td>
<td>{ multi: true }</td>
</tr>
<tr>
<td></td>
<td>} )</td>
</tr>
<tr>
<td>CREATE INDEX idx_user_id_asc ON users(user_id)</td>
<td>db.users.createIndex( { user_id: 1 } )</td>
</tr>
<tr>
<td>CREATE INDEX idx_user_id_asc_age_desc ON users(user_id, age DESC)</td>
<td>db.users.createIndex( { user_id: 1, age: -1 } )</td>
</tr>
<tr>
<td>DROP TABLE users</td>
<td>db.users.drop()</td>
</tr>
</tbody>
</table>

For more information, see db.collection.insert(), db.createCollection(), db.collection.update(), $set, $unset, db.collection.createIndex(), indexes (page 486), db.collection.drop(), and Data Modeling Concepts (page 145).

**Insert** The following table presents the various SQL statements related to inserting records into tables and the corresponding MongoDB statements.
<table>
<thead>
<tr>
<th>SQL INSERT Statements</th>
<th>MongoDB insert() Statements</th>
</tr>
</thead>
</table>
| **INSERT INTO** users(user_id,
|                                                           | db.users.insert({|                                                                        |
| age,
|                                                           |     user_id: "bcd001", age: 45, status: "A"})|                                                                        |
| status)                                                                              |                                              |
| **VALUES** ("bcd001",
|                                                           |                                              |
| 45,
|                                                           |                                              |
| "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.
<table>
<thead>
<tr>
<th>SQL SELECT Statements</th>
<th>MongoDB find() Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT *&lt;br&gt;FROM users</code></td>
<td><code>db.users.find()</code></td>
</tr>
<tr>
<td><code>SELECT id,&lt;br&gt;user_id,&lt;br&gt;status&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({&lt;br&gt;})</code></td>
</tr>
<tr>
<td><code>SELECT user_id, status&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({&lt;br&gt;})</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE status = &quot;A&quot;&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ status: &quot;A&quot; })</code></td>
</tr>
<tr>
<td><code>SELECT user_id, status&lt;br&gt;WHERE status = &quot;A&quot;&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({&lt;br&gt;status: &quot;A&quot;}&lt;br&gt;{&lt;br&gt;}&lt;br&gt;{&lt;br&gt;user_id: 1, status: 1}&lt;br&gt;{&lt;br&gt;_id: 0})</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE status != &quot;A&quot;&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ status: { $ne: &quot;A&quot; } })</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE status = &quot;A&quot; AND age = 50&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ status: &quot;A&quot;,&lt;br&gt;age: 50 })</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE status = &quot;A&quot; OR age = 50&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({&lt;br&gt;status: &quot;A&quot;,&lt;br&gt;$or: [{&lt;br&gt;status: &quot;A&quot;},&lt;br&gt;{&lt;br&gt;age: 50}]}&lt;br&gt;)</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE age &gt; 25&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ age: { $gt: 25 } })</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE age &lt; 25&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ age: { $lt: 25 } })</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE age &gt; 25 AND age &lt;= 50&lt;br&gt;FROM users</code></td>
<td><code>db.users.find({ age: { $gt: 25, $lte: 50 } })</code></td>
</tr>
<tr>
<td><code>SELECT *&lt;br&gt;WHERE user_id like &quot;%bc%</code></td>
<td><code>db.users.find( { user_id: /bc/ })</code></td>
</tr>
</tbody>
</table>
For more information, see `db.collection.find()`, `db.collection.distinct()`, `db.collection.findOne()`, `$ne`, `$and`, `$or`, `$gte`, `$lte`, `$exists`, `$lt`, `$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.

<table>
<thead>
<tr>
<th>SQL Update Statements</th>
<th>MongoDB update() Statements</th>
</tr>
</thead>
</table>
| 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.

<table>
<thead>
<tr>
<th>SQL Delete Statements</th>
<th>MongoDB remove() Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE FROM users WHERE status = &quot;D&quot;</td>
<td><code>db.users.remove({ status: &quot;D&quot; })</code></td>
</tr>
<tr>
<td>DELETE FROM users</td>
<td><code>db.users.remove({})</code></td>
</tr>
</tbody>
</table>

For more information, see `db.collection.remove()`.

### Additional Resources
- Transitioning from SQL to MongoDB (Presentation)\(^{16}\)
- Best Practices for Migrating from RDBMS to MongoDB (Webinar)\(^{17}\)
- SQL vs. MongoDB Day 1-2\(^{18}\)
- SQL vs. MongoDB Day 3-5\(^{19}\)
- MongoDB vs. SQL Day 14\(^{20}\)
- MongoDB and MySQL Compared\(^{21}\)

\(^{19}\) [http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5?jmp=docs](http://www.mongodb.com/blog/post/mongodb-vs-sql-day-3-5?jmp=docs)
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.

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

{   
  "_id" : ObjectId("51df07b094c6acd67e492f41"),
  "name" : {   
    "first" : "John",
    "last" : "McCarthy"
  }
}
```

---


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

{
    "_id" : 3,
    "name" : {
        "first" : "Grace",
        "last" : "Hopper"
    },
    "title" : "Rear Admiral",
    "birth" : ISODate("1906-12-09T05:00:00Z"),
    "death" : ISODate("1992-01-01T05:00:00Z"),
    "contribs" : [ 
        "UNIVAC",
        "compiler",
        "FLOW-MATIC",
        "COBOL"
    ],
    "awards" : [ 
        { 
            "award" : "Computer Sciences Man of the Year",
            "year" : 1969,
            "by" : "Data Processing Management Association"
        },
        { 
            "award" : "Distinguished Fellow",
            "year" : 1973,
            "by" : "British Computer Society"
        },
        { 
            "award" : "W. W. McDowell Award",
            "year" : 1976,
            "by" : "IEEE Computer Society"
        }
    ]
}
"award" : "National Medal of Technology",
"year" : 1991,
"by" : "United States"
}
)

{
"_id" : 4,
"name" : {
  "first" : "Kristen",
  "last" : "Nygaard"
},
"birth" : ISODate("1926-08-27T04:00:00Z"),
"death" : ISODate("2002-08-10T04:00:00Z"),
"contribs" : [
  "OOP",
  "Simula"
],
"awards" : [
  {
    "award" : "Rosing Prize",
    "year" : 1999,
    "by" : "Norwegian Data Association"
  },
  {
    "award" : "Turing Award",
    "year" : 2001,
    "by" : "ACM"
  },
  {
    "award" : "IEEE John von Neumann Medal",
    "year" : 2001,
    "by" : "IEEE"
  }
]
}

{
"_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",

{
    "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"
    },
    "birth" : ISODate("1966-03-06T04:00:00Z"),
    "contribs" : [
        "Scala"
    ],
    "awards" : [
        {
            "award" : "Japan Prize",
            "year" : 2011,
            "by" : "The Japan Prize Foundation"
        }
    ]
}
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 143) An introduction to data modeling in MongoDB.

Data Modeling Concepts (page 145) 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 151) Examples of possible data models that you can use to structure your MongoDB documents.

Data Model Reference (page 168) Reference material for data modeling for developers of MongoDB applications.

4.1 Data Modeling Introduction

Data in MongoDB has a flexible schema. Unlike SQL databases, where you must determine and declare a table’s schema before inserting data, MongoDB’s collections do not enforce document structure. This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of 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 171) to access the related data. Broadly, these are normalized data models.

See Normalized Data Models (page 147) 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 146) 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 148) 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.

For the MMAPv1 storage engine, if the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. When using the MMAPv1 storage engine, growth consideration can affect the decision to normalize or denormalize data. See Document Growth Considerations (page 148) for more about planning for and managing document growth for MMAPv1.
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 209). 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 148) for more information on these and other operational considerations that affect data model designs.

4.1.5 Additional Resources

- Thinking in Documents Part 1 (Blog Post)¹

4.2 Data Modeling Concepts

Consider the following aspects of data modeling in MongoDB:

Data Model Design (page 146) 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 148) Details features you should keep in mind when designing your data model, such as lifecycle management, indexing, horizontal scalability, and document growth.

GridFS (page 150) 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 143). For example data models, see Data Modeling Examples and Patterns (page 151).

¹https://www.mongodb.com/blog/post/thinking-documents-part-1?jmp=docs
4.2.1 Data Model Design

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to embed (page 146) or to use references (page 147).

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:

```
{
    _id: <ObjectId1>,
    username: "123xyz",
    contact: {
        phone: "123-456-7890",
        email: "xyz@example.com"
    },
    access: {
        level: 5,
        group: "dev"
    }
}
```

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 152).
- 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 153).

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. With the MMAPv1 storage engine, document growth can impact write performance and lead to data fragmentation.

In version 3.0.0, MongoDB uses Power of 2 Sized Allocations (page 91) as the default allocation strategy for MMAPv1 in order to account for document growth, minimizing the likelihood of data fragmentation. See Power of 2 Sized Allocations (page 91) for details. Furthermore, documents in MongoDB must be smaller than the maximum BSON document size. For bulk binary data, consider GridFS (page 150).
To interact with embedded documents, use *dot notation* to “reach into” embedded documents. See *query for data in arrays* (page 98) and *query data in embedded documents* (page 97) for more examples on accessing data in arrays and embedded documents.

**Normalized Data Models**

Normalized data models describe relationships using *references* (page 171) between documents.

```json
user document
{
   _id: <ObjectId1>,
   username: "123xyz"
}
```

```json
contact document
{
   _id: <ObjectId2>,
   user_id: <ObjectId1>,
   phone: "123-456-7890",
   email: "xyz@example.com"
}
```

```json
access document
{
   _id: <ObjectId3>,
   user_id: <ObjectId1>,
   level: 5,
   group: "dev"
}
```

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 154) for an example of referencing. For examples of various tree models using references, see *Model Tree Structures* (page 156).

**Additional Resources**

- Thinking in Documents Part 1 (Blog Post)
- Thinking in Documents (Presentation)

---

4.2.2 Operational Factors and Data Models

Modeling application data for MongoDB depends on both the data itself, as well as the characteristics of MongoDB itself. For example, different data models may allow applications to use more efficient queries, increase the throughput of insert and update operations, or distribute activity to a sharded cluster more effectively.

These factors are operational or address requirements that arise outside of the application but impact the performance of MongoDB based applications. When developing a data model, analyze all of your application’s read operations (page 60) and write operations (page 73) in conjunction with the following considerations.

Document Growth

Changed in version 3.0.0.

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.

When using the MMAPv1 storage engine, document growth can be a consideration for your data model. For MMAPv1, if the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. With MongoDB 3.0.0, however, the default use of the Power of 2 Sized Allocations (page 91) minimizes the occurrences of such re-allocations as well as allows for the effective reuse of the freed record space.

When using MMAPv1, if your applications require updates that will frequently cause document growth to exceeds the current power of 2 allocation, you may want to refactor your data model to use references between data in distinct documents rather than a denormalized data model.

You may also use a pre-allocation strategy to explicitly avoid document growth. Refer to the Pre-Aggregated Reports Use Case7 for an example of the pre-allocation approach to handling document growth.

See Storage (page 89) 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. 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 164) for an example data model that provides atomic updates for a single document.

---

6https://www.mongodb.com/products/consulting?jmp=docs#rapid_start
7https://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports
8Document-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.
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 674). Selecting the proper *shard key* (page 674) 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 661) and *Shard Keys* (page 674) 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 546) for more information on indexes as well as *Analyze Query Performance* (page 111). Additionally, the MongoDB *database profiler* (page 227) 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:

```json
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ...}
```

If the total number of documents is low, you may group documents into collection by type. For logs, consider maintaining distinct log collections, such as *logs_dev* and *logs_debug*. The *logs_dev* collection would contain only the documents related to the dev environment.

Generally, having 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 using the **mmapv1** storage engine 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 211) 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 209). 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

*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, ⁹ 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.

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 749).

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.

---

⁹ The use of the term `chunks` in the context of GridFS is not related to the use of the term `chunks` in the context of sharding.
GridFS Collections

GridFS stores files in two collections:

- chunks stores the binary chunks. For details, see The chunks Collection (page 174).
- files stores the file’s metadata. For details, see The files Collection (page 175).

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 174).

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 174).

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.createIndex( { files_id: 1, n: 1 }, { unique: true } );

Additional Resources

- Building MongoDB Applications with Binary Files Using GridFS: Part 1
- Building MongoDB Applications with Binary Files Using GridFS: Part 2

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 152)  Examples for modeling relationships between documents.

Model One-to-One Relationships with Embedded Documents (page 152)  Presents a data model that uses embedded documents (page 146) to describe one-to-one relationships between connected data.
Model One-to-Many Relationships with Embedded Documents (page 153) Presents a data model that uses embedded documents (page 146) to describe one-to-many relationships between connected data.

Model One-to-Many Relationships with Document References (page 154) Presents a data model that uses references (page 147) to describe one-to-many relationships between documents.

Model Tree Structures (page 156) Examples for modeling tree structures.

Model Tree Structures with Parent References (page 157) Presents a data model that organizes documents in a tree-like structure by storing references (page 147) to “parent” nodes in “child” nodes.

Model Tree Structures with Child References (page 158) Presents a data model that organizes documents in a tree-like structure by storing references (page 147) to “child” nodes in “parent” nodes.

See Model Tree Structures (page 156) for additional examples of data models for tree structures.

Model Specific Application Contexts (page 164) Examples for models for specific application contexts.

Model Data for Atomic Operations (page 164) 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 165) 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 152) Presents a data model that uses embedded documents (page 146) to describe one-to-one relationships between connected data.

Model One-to-Many Relationships with Embedded Documents (page 153) Presents a data model that uses embedded documents (page 146) to describe one-to-many relationships between connected data.

Model One-to-Many Relationships with Document References (page 154) Presents a data model that uses references (page 147) to describe one-to-many relationships between documents.

Model One-to-One Relationships with Embedded Documents

Overview

Data in MongoDB has a flexible schema. Collections do not enforce document structure. Decisions that affect how you model data can affect application performance and database capacity. See Data Modeling Concepts (page 145) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses embedded (page 146) 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.
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:

```json
{
  _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**

**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 145) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses embedded (page 146) 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.

```json
{
  _id: "joe",
  name: "Joe Bookreader"
}
{
  patron_id: "joe",
  street: "123 Fake Street",
  city: "Faketon",
  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:

```json
{
    _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**

**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 145) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses references (page 147) 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:

```json
{
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirollf" ],
    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:

```json
{
    name: "O'Reilly Media",
    founded: 1980,
    location: "CA",
    books: [12346789, 234567890, ...]
}

{
    _id: 12346789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirollf" ],
    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,
}
```
To avoid mutable, growing arrays, store the publisher reference inside the book document:

```json

    {
        _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 157]) Presents a data model that organizes documents in a tree-like structure by storing references (page 147) to “parent” nodes in “child” nodes.

**Model Tree Structures with Child References** ([page 158]) Presents a data model that organizes documents in a tree-like structure by storing references (page 147) to “child” nodes in “parent” nodes.

**Model Tree Structures with an Array of Ancestors** ([page 160]) Presents a data model that organizes documents in a tree-like structure by storing references (page 147) to “parent” nodes and an array that stores all ancestors.

**Model Tree Structures with Materialized Paths** ([page 161]) 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 163]) 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

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 145) 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 147) 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:

```javascript
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" } )
```
The query to retrieve the parent of a node is fast and straightforward:

```javascript
db.categories.findOne( { _id: "MongoDB" } ).parent
```

- You can create an index on the field `parent` to enable fast search by the parent node:

```javascript
db.categories.createIndex( { parent: 1 } )
```

- You can query by the `parent` field to find its immediate children nodes:

```javascript
db.categories.find( { parent: "Databases" } )
```

The **Parent Links** pattern provides a simple solution to tree storage but requires multiple queries to retrieve subtrees.

### Model Tree Structures with Child References

#### Overview

Data in MongoDB has a flexible schema. **Collections** do not enforce document structure. Decisions that affect how you model data can affect application performance and database capacity. See **Data Modeling Concepts** (page 145) 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 147) 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:

```
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" ] })
```

- 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.createIndex({ 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**

**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 145) 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 147) 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:

```javascript
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "Databases" } )
db.categories.insert( { _id: "dbm", ancestors: [ "Books", "Programming", "Databases" ], parent: "Databases" } )
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:
  ```javascript
db.categories.findOne( { _id: "MongoDB" } ).ancestors
```

- You can create an index on the field *ancestors* to enable fast search by the ancestors nodes:
  ```javascript
db.categories.createIndex( { ancestors: 1 } )
```

- You can query by the field *ancestors* to find all its descendants:
  ```javascript
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 161) pattern but is more straightforward to use.

### Model Tree Structures with Materialized Paths

#### Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 145) 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:

```javascript
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: "dbm", path: ",Books,Programming,Databases," } )
```
• You can query to retrieve the whole tree, sorting by the field `path`:

```javascript
db.categories.find().sort( { path: 1 } )
```

• You can use regular expressions on the `path` field to find the descendants of `Programming`:

```javascript
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:

```javascript
db.categories.find( { path: /^,Books,/ } )
```

• To create an index on the field `path` use the following invocation:

```javascript
db.categories.createIndex( { 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

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 145) 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:

4.3. Data Model Examples and Patterns 163
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 164)* 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 165)* 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 166)* Describes two methods to model monetary data in MongoDB.

**Model Time Data** *(page 168)* Describes how to deal with local time in MongoDB.

#### Model Data for Atomic Operations

**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.

```json
{  
    _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:

```javascript
db.books.update({  
    _id: 123456789,  
    available: { $gt: 0 }  
},  
{  
    $inc: { available: -1 },  
    $push: { checkout: { by: "abc", date: new Date() } }  
})
```

164 Chapter 4. Data Models
The operation returns a `WriteResult()` object that contains information on the status of the operation:

```javascript
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

**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 166) section for more information.

In 2.4, MongoDB provides a text search feature. See Text Indexes (page 500) 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:

```javascript
{ 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:

```javascript
db.volumes.createIndex( { 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:

```javascript
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

4.3. Data Model Examples and Patterns 165
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

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 167) which multiplies the monetary value by a power of 10.

• *Arbitrary Precision* (page 167) 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 167).

• If you need to be able to do server-side arithmetic, e.g., $inc, $mul, and aggregation framework arithmetic, use *Exact Precision* (page 167).
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 167).
- If server-side approximations are sufficient, possibly with client-side post-processing, see *Arbitrary Precision* (page 167).

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.

```json
{ 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 166).

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:

```json
{   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 167).
Model Time Data

Overview

MongoDB stores times in UTC (page 180) 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.

```javascript
var now = new Date();
db.data.save( { date: now, offset: now.getTimezoneOffset() } );
```

You can reconstruct the original local time by applying the saved offset:

```javascript
var record = db.data.findOne();
var localNow = new Date( record.date.getTime() - ( record.offset * 60000 ) );
```

4.4 Data Model Reference

Documents (page 168) MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs.

Database References (page 171) Discusses manual references and DBRefs, which MongoDB can use to represent relationships between documents.

GridFS Reference (page 174) Convention for storing large files in a MongoDB Database.

ObjectId (page 176) 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 178) Outlines the unique BSON types used by MongoDB. See BSONspec.org for the complete BSON specification.

4.4.1 Documents

MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs:

```json
{ "item": "pencil", "qty": 500, "type": "no.2" }
```

Most user-accessible data structures in MongoDB are documents, including:

- All database records.
- Query selectors (page 60), which define what records to select for read, update, and delete operations.
- Update definitions (page 73), which define what fields to modify during an update.
- Index specifications (page 486), which define what fields to index.
- Data output by MongoDB for reporting and configuration, such as the output of the serverStatus and the replica set configuration document (page 652).

12http://bsonspec.org/
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. See also BSON Types (page 178).

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:

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

The value of a field can be any of the BSON data types (page 178), including other documents, arrays, and arrays of documents. The following document contains values of varying types:

```javascript
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 168) have the following restrictions on field names:

- The field name _id is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
- The field names cannot start with the dollar sign ($) character.
- The field names cannot contain the dot (.) character.
- The field names cannot contain the null character.

---

13http://bsonspec.org/
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 178), 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 176).

- 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 125).

• 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\textsuperscript{14} 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:

'\langle array\rangle.<\langle index\rangle>'

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:

'\langle embedded document\rangle.<\langle field\rangle>'

See also:

• *Embedded Documents* (page 97) for dot notation examples with embedded documents.

• *Arrays* (page 98) for dot notation examples with arrays.

**Additional Resources**

• Thinking in Documents Part 1 (Blog Post)\textsuperscript{15}

**4.4.2 Database References**

MongoDB does not support joins. In MongoDB some data is *denormalized*, or stored with related data in *documents* to remove the need for joins. However, in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

MongoDB applications use one of two methods for relating documents:

• *Manual references* (page 172) 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.

---

\textsuperscript{14}https://api.mongodb.org/

\textsuperscript{15}https://www.mongodb.com/blog/post/thinking-documents-part-1?jmp=docs
DBRefs (page 173) 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 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:

```javascript
original_id = ObjectId()

db.places.insert({
   "_id": original_id,
   "name": "Broadway Center",
   "url": "bc.example.net"
})

db.people.insert({
   "name": "Erin",
   "places_id": original_id,
   "url": "bc.example.net/Erin"
})
```

Then, when a query returns the document from the people collection you can, if needed, make a second query for the document referenced by the places_id field in the places collection.

Use

For nearly every case where you want to store a relationship between two documents, use manual references (page 172). 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.

---

16 Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.
DBRefs

Background

DBRefs are a convention for representing a document, rather than a specific reference type. They include the name of the collection, and in some cases the database 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 class and the FetchDBRef method.

4.4. Data Model Reference 173
Haskell  The Haskell driver contains no support for DBRefs. You can traverse references manually.

Java  The `DBRef`\(^{19}\) 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`\(^{20}\) class and the `dereference`\(^{21}\) method.

Perl  The Perl driver contains no support for DBRefs. You can traverse references manually or use the `MongoDBx::AutoDeref`\(^{22}\) CPAN module.

PHP  The PHP driver supports DBRefs, including the optional \$_db reference, using the `MongoDBRef`\(^{23}\) class.

Python  The Python driver supports DBRefs using the `DBRef`\(^{24}\) class and the `dereference`\(^{25}\) method.

Ruby  The Ruby driver supports DBRefs using the `DBRef`\(^{26}\) class and the `dereference`\(^{27}\) 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 172) 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

`GridFS` stores files in two collections:

- `chunks` stores the binary chunks. For details, see `The chunks Collection` (page 174).
- `files` stores the file’s metadata. For details, see `The files Collection` (page 175).

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

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

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

See also:

`GridFS` (page 150) 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:\n
---

\(^{19}\)https://api.mongodb.org/java/current/com/mongodb/DBRef.html

\(^{20}\)http://mongodb.github.io/node-mongodb-native/api-bson-generated/db_ref.html

\(^{21}\)http://mongodb.github.io/node-mongodb-native/api-generated/db.html#dereference

\(^{22}\)http://search.cpan.org/dist/MongoDBx-AutoDeref/


\(^{24}\)https://api.mongodb.org/python/current/api/bson/dbref.html

\(^{25}\)https://api.mongodb.org/python/current/api/pymongo/database.html#pymongo.database.Database.dereference

\(^{26}\)https://api.mongodb.org/ruby/current/BSON/DBRef.html

\(^{27}\)https://api.mongodb.org/ruby/current/Mongo/DB.html#dereference-instance_method
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 151).

### 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:

```json
{
   "\_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

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 168) 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:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
ObjectId("507f191e810c19729de860ea").str
```
This operation will return the following hexadecimal string:

507f191e810c19729de860ea

To return the hexadecimal string representation of an ObjectId() object, use the `valueOf()` method as follows:

```javascript
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(...)`) uses the `toString()` method as follows:

```javascript
ObjectId("507f191e810c19729de860ea").toString()
```

This operation will return the following string output:

`ObjectId("507f191e810c19729de860ea")`

### 4.4.5 BSON Types

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](http://bsonspec.org).

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Binary data</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Undefined</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Object id</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Regular Expression</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>JavaScript</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Symbol</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>JavaScript (with scope)</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>32-bit integer</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>64-bit integer</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Min key</td>
<td>255</td>
<td>Query with -1.</td>
</tr>
<tr>
<td>Max key</td>
<td>127</td>
<td></td>
</tr>
</tbody>
</table>

To determine a field’s type, see *Check Types in the mongo Shell* (page 269).

If you convert BSON to JSON, see the Extended JSON reference.

---

28http://bsonspec.org/
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
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Changed in version 3.0.0: Date objects sort before Timestamp objects. Previously Date and Timestamp objects sorted together.

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 176) 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.\(^{29}\) In addition, MongoDB $\textit{regex}$ queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for \textit{internal} MongoDB use and is \textbf{not} associated with the regular \textit{Date} (page 180) type. Timestamp values are a 64 bit value where:

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

Within a single \texttt{mongod} instance, timestamp values are always unique.

In replication, the \texttt{oplog} has a \texttt{ts} field. The values in this field reflect the operation time, which uses a BSON timestamp value.

\textbf{Note:} The BSON timestamp type is for \textit{internal} MongoDB use. For most cases, in application development, you will want to use the BSON date type. See \textit{Date} (page 180) 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:

```javascript
var a = new Timestamp();
db.test.insert( { ts: a } );
```

Then, the \texttt{db.test.find()} operation will return a document that resembles the following:

```json
{ "_id" : ObjectId("542c2b97bac0595474108b48"), "ts" : Timestamp(1412180887, 1) }
```

If \texttt{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 \texttt{_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 \textit{official BSON specification}\(^{30}\) refers to the BSON Date type as the \textit{UTC datetime}.

Changed in version 2.0: BSON Date type is signed.\(^{31}\) Negative values represent dates before 1970.

Example

Construct a Date using the \texttt{new Date()} constructor in the \texttt{mongo} shell:

\(^{29}\) Given strings using UTF-8 character sets, using \texttt{sort()} on strings will be reasonably correct. However, because internally \texttt{sort()} uses the \texttt{C++} \texttt{strcmp} api, the sort order may handle some characters incorrectly.

\(^{30}\) \url{http://bsonspec.org/#/specification}

\(^{31}\) Prior to version 2.0, \texttt{Date} values were incorrectly interpreted as \textit{unsigned} integers, which affected sorts, range queries, and indexes on \texttt{Date} fields. Because indexes are not recreated when upgrading, please re-index if you created an index on \texttt{Date} values with an earlier version, and dates before 1970 are relevant to your application.
```javascript
var mydate1 = new Date()

Example
Construct a Date using the ISODate() constructor in the mongo shell:

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

Example
Return the Date value as string:
```javascript
mydate1.toString()
```

Example
Return the month portion of the Date value; months are zero-indexed, so that January is month 0:
```javascript
mydate1.getMonth()
```
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 183) Core conceptual documentation of operational practices for managing MongoDB deployments and systems.

  MongoDB Backup Methods (page 184) Describes approaches and considerations for backing up a MongoDB database.

  Monitoring for MongoDB (page 187) An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

  Production Notes (page 197) A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.

Continue reading from Administration Concepts (page 183) for additional documentation of MongoDB administration.

Administration Tutorials (page 220) Tutorials that describe common administrative procedures and practices for operations for MongoDB instances and deployments.

  Configuration, Maintenance, and Analysis (page 221) Describes routine management operations, including configuration and performance analysis.

  Backup and Recovery (page 243) Outlines procedures for data backup and restoration with mongod instances and deployments.

Continue reading from Administration Tutorials (page 220) for more tutorials of common MongoDB maintenance operations.

Administration Reference (page 286) 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 602) and Sharded Cluster Tutorials (page 690) 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 184) Higher level documentation of key concepts for the operation and maintenance of MongoDB deployments.

MongoDB Backup Methods (page 184) Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB (page 187) An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration (page 192) Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Continue reading from Operational Strategies (page 184) for additional documentation.

Data Management (page 207) Core documentation that addresses issues in data management, organization, maintenance, and lifecycle management.

Data Center Awareness (page 207) 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 209) 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 211) TTL collections make it possible to automatically remove data from a collection based on the value of a timestamp and are useful for managing data like machine generated event data that are only useful for a limited period of time.

Optimization Strategies for MongoDB (page 213) Techniques for optimizing application performance with MongoDB.

Continue reading from Optimization Strategies for MongoDB (page 213) for additional documentation.

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 184) Describes approaches and considerations for backing up a MongoDB database.

Monitoring for MongoDB (page 187) An overview of monitoring tools, diagnostic strategies, and approaches to monitoring replica sets and sharded clusters.

Run-time Database Configuration (page 192) Outlines common MongoDB configurations and examples of best-practice configurations for common use cases.

Production Notes (page 197) A collection of notes that describe best practices and considerations for the operations of MongoDB instances and deployments.

MongoDB Backup Methods

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 185)
- Backup a Database with mongodump (page 250)
- MongoDB Cloud Manager Backup (page 186)
• Ops Manager Backup Software (page 186)

**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 243) and Backup a Sharded Cluster with Filesystem Snapshots (page 256) for complete instructions on using LVM to create snapshots. Also see Back up and Restore Processes for MongoDB on Amazon EC21.

**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 652).

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

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

To mitigate the impact of mongodump on the performance of the replica set, use mongodump to capture backups from a secondary (page 567) 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.

---

1https://docs.mongodb.org/ecosystem/tutorial/backup-and-restore-mongod-on-amazon-ec2
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 249), *Backup a Small Sharded Cluster with mongodump* (page 255), and *Backup a Sharded Cluster with Database Dumps* (page 258) 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 186) 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 243) 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 247) Describes procedure for restoring a replica set from an archived backup such as a `mongodump` or MongoDB Cloud Manager\(^7\) Backup file.

*Back Up and Restore with MongoDB Tools* (page 249) Describes a procedure for exporting the contents of a database to either a binary dump or a textual exchange format, and for importing these files into a database.

*Backup and Restore Sharded Clusters* (page 254) Detailed procedures and considerations for backing up sharded clusters and single shards.

*Recover Data after an Unexpected Shutdown* (page 263) Recover data from MongoDB data files that were not properly closed or have an invalid state.

---

\(^2\)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
Additional Resources

- Backup and it’s Role in Disaster Recovery White Paper
- Backup vs. Replication: Why Do You Need Both?
- MongoDB Production Readiness Consulting Package

Monitoring for MongoDB

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

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

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

Monitoring Strategies

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

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

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

MongoDB Reporting Tools

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

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

---

5.1. Administration Concepts 187
**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\(^\text{17}\) and Ops Manager\(^\text{18}\). Nevertheless, all administrators should be familiar with the data provided by `serverStatus`.

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

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

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

\(^{17}\)https://cloud.mongodb.com/?jmp=docs
\(^{18}\)https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
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.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ganglia</td>
<td>mongodb-ganglia</td>
<td>Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections. Parses output from the <code>serverStatus</code> and <code>replSetGetStatus</code> commands.</td>
</tr>
<tr>
<td>Ganglia</td>
<td>gmond_python_modules</td>
<td>Realtime monitoring tool for MongoDB servers. Shows current operations ordered by durations every second.</td>
</tr>
<tr>
<td>Motop</td>
<td>None</td>
<td>A top like tool. Retrieves server statistics.</td>
</tr>
<tr>
<td>mtop</td>
<td>None</td>
<td>Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).</td>
</tr>
<tr>
<td>Munin</td>
<td>mongo-munin</td>
<td>Some additional munin plugins not in the main distribution.</td>
</tr>
<tr>
<td>Munin</td>
<td>mongomon</td>
<td>A simple Nagios check script, written in Python.</td>
</tr>
<tr>
<td>Munin</td>
<td>munin-plugins Ubuntu PPA</td>
<td>Also consider dex, an index and query analyzing tool for MongoDB that compares MongoDB log files and indexes to make indexing recommendations.</td>
</tr>
<tr>
<td>Nagios</td>
<td>nagios-plugin-mongodb</td>
<td>See also: Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced.</td>
</tr>
</tbody>
</table>

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

---

20[https://github.com/quiiver/mongodb-ganglia](https://github.com/quiiver/mongodb-ganglia)
21[https://github.com/ganglia/gmond_python_modules](https://github.com/ganglia/gmond_python_modules)
22[https://github.com/tart/motop](https://github.com/tart/motop)
23[https://github.com/beaufour/mtop](https://github.com/beaufour/mtop)
25[https://github.com/erh/mongo-munin](https://github.com/erh/mongo-munin)
26[https://github.com/pcdummy/mongomon](https://github.com/pcdummy/mongomon)
27[https://launchpad.net/chris-lea/+archive/munin-plugins](https://launchpad.net/chris-lea/+archive/munin-plugins)
29[https://github.com/mzupan/nagios-plugin-mongodb](https://github.com/mzupan/nagios-plugin-mongodb)
30[https://github.com/mongolab/dex](https://github.com/mongolab/dex)

---

5.1. Administration Concepts 189
### Name | Notes
--- | ---
**MongoDB Cloud Manager**[^32] | MongoDB Cloud Manager is a cloud-based suite of services for managing MongoDB deployments. MongoDB Cloud Manager provides monitoring, backup, and automation functionality. For an on-premise solution, see also Ops Manager, available in MongoDB Enterprise Advanced[^33].
**Scout[^34]** | Several plugins, including MongoDB Monitoring[^35], MongoDB Slow Queries[^36], and MongoDB Replica Set Monitoring[^37].
**Server Density[^38]** | Dashboard for MongoDB[^39], MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.
**Application Performance Management[^40]** | IBM has an Application Performance Management SaaS offering that includes monitor for MongoDB and other applications and middleware.

### Process Logging

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

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

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

For example:

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

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

The following *database commands* also affect logging:

- **getLog.** Displays recent messages from the `mongod` process log.
- **logRotate.** Rotates the log files for `mongod` processes only. See *Rotate Log Files* (page 231).

### Diagnosing Performance Issues

As you develop and operate applications with MongoDB, you may want to analyze the performance of the database as the application. *Analyzing MongoDB Performance* (page 213) discusses some of the operational factors that can influence performance.

[^32]: [https://cloud.mongodb.com/?jmp=docs](https://cloud.mongodb.com/?jmp=docs)
[^34]: [http://scoutapp.com](http://scoutapp.com)
[^38]: [http://www.serverdensity.com](http://www.serverdensity.com)
[^40]: [http://ibmserviceengage.com](http://ibmserviceengage.com)
Replication and Monitoring

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

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

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

By default, the oplog is 5 percent of total available disk space on 64-bit systems. For more information about changing the oplog size, see the Change the Size of the Oplog (page 628)

For causes of replication lag, see Replication Lag (page 647).

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 667) documentation for more information.

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

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

MongoDB Cloud Manager[^41] and Ops Manager[^42] monitor config servers and can create notifications if a config server

[^41]: https://cloud.mongodb.com/?jmp=docs
[^42]: https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
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

Run-time Database Configuration

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

While both interfaces provide access to the same collection of options and settings, this document primarily uses the configuration file interface. If you run MongoDB using a control script or installed from a package for your operating system, you likely already have a configuration file located at `/etc/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.

---

43https://docs.cloud.mongodb.com/
44https://docs.opsmanager.mongodb.com/current/application
45https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness
Configure the Database

Consider the following basic configuration which uses the YAML format:

```yaml
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 19) and .rpm (page 7) 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 configuration file format:
```
bind_ip = 127.0.0.1,10.8.0.10,192.168.4.24
auth = true
```

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 320)

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:
```
replSet = set0
```

---

46 http://docs.mongodb.org/v2.4/reference/configuration-options
47 http://docs.mongodb.org/v2.4/reference/configuration-options
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

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

Or, if using the older configuration file format:

```text
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 323) section for information on configuring authentication with replica sets.

The Replication (page 559) document for more information on replication in MongoDB and replica set configuration in general.

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

Note: Config servers are not replica sets.

To set up one or three “config server” instances as normal mongod instances, and then add the following configuration option:

In YAML format

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

Or, if using the older configuration file format:

```text
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, with the shardsvr value for the sharding.clusterRole setting:

---

48http://docs.mongodb.org/v2.4/reference/configuration-options
49http://docs.mongodb.org/v2.4/reference/configuration-options
sharding:
  clusterRole: shardsvr

Or, if using the older configuration file format:\footnote{http://docs.mongodb.org/v2.4/reference/configuration-options}

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:\footnote{http://docs.mongodb.org/v2.4/reference/configuration-options}

```
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,\footnote{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.} and ensure this setting is consistent among all mongos instances.

See also:

The *Sharding* (page 661) 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 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 193) 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:\footnote{http://docs.mongodb.org/v2.4/reference/configuration-options}

```
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 215) 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 215). 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.

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

For more information, see also *Database Profiling* (page 215) and *Analyzing MongoDB Performance* (page 213).

**Production Notes**

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

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

**MongoDB**

**Storage Engines**  Changed in version 3.0: MongoDB includes support for two storage engines: MMAPv1 (page 91), the storage engine available in previous versions of MongoDB, and WiredTiger (page 90). MongoDB uses the MMAPv1 engine by default.

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

**Supported Platforms**  MongoDB distributions are currently available for Mac OS X, Linux, Windows Server 2012, Windows Server 2008 R2 64bit, Windows 7 (64 bit), Windows Vista, and Solaris. The MongoDB distribution for Solaris does not include support for the WiredTiger storage engine (page 90).
For a full list of the recommended operating systems for production deployments, see: *Recommended Operating Systems for Production Deployments* (page 5).

See also:

*Platform Specific Considerations* (page 202)

**Use the Latest Stable Packages**  Be sure you have the latest stable release.

All releases are available on the [Downloads](http://www.mongodb.org/downloads) page. The [Downloads](http://www.mongodb.org/downloads) page is a good place to verify the current stable release, even if you are installing via a package manager.

**Use 64-bit Builds**  Always use 64-bit builds for production.

Although the 32-bit builds exist, they are *unsuitable* for production deployments. 32-bit builds also do not support the WiredTiger storage engine. For more information, see the *32-bit limitations page* (page 746)

**Concurrency**

**MMAPv1**  Changed in version 3.0: Beginning with MongoDB 3.0, **MMAPv1** (page 91) provides *collection-level locking*: All collections have a unique readers-writer lock that allows multiple clients to modify documents in different collections at the same time.

For MongoDB versions 2.2 through 2.6 series, each database has a readers-writer lock that allows concurrent read access to a database, but gives exclusive access to a single write operation per database. See the *Concurrency* (page 758) page for more information. In earlier versions of MongoDB, all write operations contended for a single readers-writer lock for the entire *mongod* instance.

**WiredTiger**  *WiredTiger* (page 90) supports concurrent access by readers and writers to the documents in a collection. Clients can read documents while write operations are in progress, and multiple threads can modify different documents in a collection at the same time.

See also:

*Allocate Sufficient RAM and CPU* (page 200)

**Data Consistency**

**Journaling**  MongoDB uses *write ahead logging* to an on-disk *journal*. Journaling guarantees that MongoDB can quickly recover *write operations* (page 73) that were not written to data files in cases where *mongod* terminated as a result of a crash or other serious failure.

Leave journaling enabled in order to ensure that *mongod* will be able to recover its data files and keep the data files in a valid state following a crash. See *Journaling* (page 308) for more information.

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

---

59http://www.mongodb.org/downloads
60http://www.mongodb.org/downloads
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 78) document for more information about choosing an appropriate write concern level for your deployment.

Networking

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

Note: By default, authorization (page 324) is not enabled, and mongod assumes a trusted environment. Enable authorization mode as needed. For more information on authentication mechanisms supported in MongoDB as well as authorization in MongoDB, see Authentication (page 320) and Authorization (page 324).

For additional information and considerations on security, refer to the documents in the Security Section (page 317), specifically:

- Security Checklist (page 431)
- Configuration Options (page 327)
- Firewalls (page 328)
- Network Security Tutorials (page 335)

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

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

See HTTP Status Interface (page 329).

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

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

See also Allocate Sufficient RAM and CPU (page 200).

Hardware Considerations

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

Allocate Sufficient RAM and CPU

**MMAPv1** The MMAPv1 storage engine is not CPU bound due to its concurrency model. As such, increasing the number of cores can help but does not provide significant return.

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

**WiredTiger** The WiredTiger storage engine is CPU bound. Specifically, the total number of active threads (i.e. concurrent operations) relative to the number of CPUs can impact performance:

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

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

WiredTiger’s default configuration will use either 1GB or half of the installed physical RAM for cache, whichever is larger. This size is tunable through the `storage.wiredTiger.engineConfig.cacheSizeGB` setting, and should be large enough to hold your entire working set.

The default `storage.wiredTiger.engineConfig.cacheSizeGB` setting assumes that there is a single `mongod` instance per node. If a single node contains multiple instances, then you should adjust the `storage.wiredTiger.engineConfig.cacheSizeGB` setting to accommodate the working set as well as the other `mongod` instances.

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

See also:

*Concurrency* (page 198)

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

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

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

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

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

See also:
• The MySQL “swap insanity” problem and the effects of NUMA post, which describes the effects of NUMA on databases. The post introduces NUMA and its goals, and illustrates how these goals are not compatible with production databases. Although the blog post addresses the impact of NUMA for MySQL, the issues for MongoDB are similar.

• NUMA: An Overview.

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 670); 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/*.

Disk and Storage Systems

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

For the MMAPv1 storage engine, the method mongod uses to map files to memory ensures that the operating system will never store MongoDB data in swap space. On Windows systems, using MMAPv1 requires extra swap space due to commitment limits. For details, see MongoDB on Windows (page 204).

The WiredTiger storage engine also ensures that the operating system does not store MongoDB data in swap space.

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

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

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

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

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

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

---

63https://queue.acm.org/detail.cfm?id=2513149
64http://www.kernel.org/doc/Documentation/sysctl/vm.txt
Separate Components onto Different Storage Devices  For improved performance, consider separating your database’s data, journal, and logs onto different storage devices, based on your application’s access and write pattern.

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

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

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

Architecture

Replica Sets  See the `Replica Set Architectures` (page 572) document for an overview of architectural considerations for replica set deployments.

Sharded Clusters  See the `Sharded Cluster Production Architecture` (page 672) document for an overview of recommended sharded cluster architectures for production deployments.

See also:

`Design Notes` (page 218)

Compression

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

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

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

Platform Specific Considerations

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

MongoDB on Linux

\(^65\)http://www.gnu.org/software/libc/
Kernel and File Systems  When running MongoDB in production on Linux, it is recommended that you use Linux kernel version 2.6.36 or later.

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

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

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

<table>
<thead>
<tr>
<th>Linux Distribution</th>
<th>Filesystem</th>
<th>Kernel Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS 5.5</td>
<td>ext4, xfs</td>
<td>2.6.18-194.el5</td>
</tr>
<tr>
<td>CentOS 5.6</td>
<td>ext4, xfs</td>
<td>2.6.18-3.0.el5</td>
</tr>
<tr>
<td>CentOS 5.8</td>
<td>ext4, xfs</td>
<td>2.6.18-308.82.el5</td>
</tr>
<tr>
<td>CentOS 6.1</td>
<td>ext4, xfs</td>
<td>2.6.32-131.0.15.el6.x86_64</td>
</tr>
<tr>
<td>RHEL 5.6</td>
<td>ext4</td>
<td>2.6.18-3.0</td>
</tr>
<tr>
<td>RHEL 6.0</td>
<td>xfs</td>
<td>2.6.32-71</td>
</tr>
<tr>
<td>Ubuntu 10.04.4 LTS</td>
<td>ext4, xfs</td>
<td>2.6.32-38-server</td>
</tr>
<tr>
<td>Amazon Linux AMI release 2012.03</td>
<td>ext4</td>
<td>3.2.12-3.2.4.amzn1.x86_64</td>
</tr>
</tbody>
</table>

**fsync() on Directories**

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

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

- Turn off `atime` for the storage volume containing the database files.
- Set the file descriptor limit, `-n`, and the user process limit (ulimit), `-u`, above 20,000, according to the suggestions in the `ulimit` (page 287) 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 221).
- Disable NUMA in your BIOS. If that is not possible, see MongoDB on NUMA Hardware (page 200).
- Configure SELinux on Red Hat. For more information, see Configure SELinux for MongoDB (page 9) and Configure SELinux for MongoDB Enterprise (page 35).

For the MMAPv1 storage engine:

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

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

For all MongoDB deployments:

- Use the Network Time Protocol (NTP) to synchronize time among your hosts. This is especially important in sharded clusters.
**MongoDB and TLS/SSL Libraries**  On Linux platforms, you may observe one of the following statements in the MongoDB log:

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

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:

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

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

```plaintext
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:

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

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

---

**MongoDB on Windows**

**MongoDB Using MMAPv1**

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

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

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

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

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

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

---

66http://support.microsoft.com/kb/2731284
**MongoDB 3.0 Using WiredTiger**  For MongoDB instances using the WiredTiger storage engine, performance on Windows is comparable to performance on Linux.

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

For all platforms, consider *Scheduling for Virtual Devices* (page 202).

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

**Azure**  For all MongoDB deployments using Azure, you **must** mount the volume that hosts the mongod instance’s dbPath with the *Host Cache Preference READ/WRITE*. This applies to all Azure deployments, using any guest operating system.

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

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

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

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

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

**On Linux systems:**

- To view the keep alive setting, you can use one of the following commands:
  ```
sysctl net.ipv4.tcp_keepalive_time
  
  Or:
  
cat /proc/sys/net/ipv4/tcp_keepalive_time
  ```

  The value is measured in seconds.

- To change the *tcp_keepalive_time* value, you can use one of the following command:
  ```
sudo sysctl -w net.ipv4.tcp_keepalive_time=<value>
  
  Or:
  
echo <value> | sudo tee /proc/sys/net/ipv4/tcp_keepalive_time
  ```

  These operations do not persist across system reboots. To persist the setting, add the following line to /etc/sysctl.conf:
  ```
  net.ipv4.tcp_keepalive_time = <value>
  ```

[^67]: https://cloud.mongodb.com/?jmp=docs
[^68]: https://docs.cloud.mongodb.com/tutorial/configure-aws-settings/
On Linux, `mongod` and `mongos` processes limit the keepalive to a maximum of 300 seconds (5 minutes) on their own sockets by overriding keepalive values greater than 5 minutes.

For Windows systems:

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

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

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

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

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

  Windows users should consider the [Windows Server Technet Article on KeepAliveTime](https://technet.microsoft.com/en-us/library/cc957549.aspx) for more information on setting keep alive for MongoDB deployments on Windows systems.

VMWare  MongoDB is compatible with VMWare.

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

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

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

MongoDB on Solaris  The MongoDB distribution for Solaris does not include support for the [WiredTiger storage engine](page 90).

Performance Monitoring

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

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

```shell
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` 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.

---


70[http://www.gropp.org/?id=projects&sub=bwm-ng](http://www.gropp.org/?id=projects&sub=bwm-ng)
Backups

To make backups of your MongoDB database, please refer to *MongoDB Backup Methods Overview* (page 184).

Additional Resources

- Blog Post: Capacity Planning and Hardware Provisioning for MongoDB In Ten Minutes
- Whitepaper: MongoDB Multi-Data Center Deployments
- Whitepaper: Security Architecture
- Whitepaper: MongoDB Architecture Guide
- Presentation: MongoDB Administration 101
- MongoDB Production Readiness Consulting Package

5.1.2 Data Management

These documents 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 207)  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 209)  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 211)  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.

Data Center Awareness

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

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

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

**Operational Segregation in MongoDB Deployments** (page 208)  MongoDB lets you specify that certain application operations use certain mongod instances.

**Tag Aware Sharding** (page 682)  Tags associate specific ranges of shard key values with specific shards for use in managing deployment patterns.

---

74https://www.mongodb.com/lp/whitepaper/architecture-guide?jmp=docs
75http://www.mongodb.com/presentations/webinar-mongodb-administration-101?jmp=docs
76https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness
Operational Segregation in MongoDB Deployments

Operational Overview MongoDB includes a number of features that allow database administrators and developers to segregate application operations to MongoDB deployments by functional or geographical groupings. This capability provides “data center awareness,” which allows applications to target MongoDB deployments with consideration of the physical location of the mongod instances. MongoDB supports segmentation of operations across different dimensions, which may include multiple data centers and geographical regions in multi-data center deployments, racks, networks, or power circuits in single data center deployments.

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

Specifically, with MongoDB, you can:

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

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

- Read Preferences (page 588), which controls how drivers help applications target read operations to members of a replica set.
- Write Concerns (page 78), which controls how MongoDB ensures that write operations propagate to members of a replica set.
- Replica Set Tags (page 635), 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 682), 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 559), and sharding (page 661).

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments
- Webinar: Multi-Data Center Deployment

Further Reading

- The Write Concern (page 78) and Read Preference (page 588) documents, which address capabilities related to data center awareness.

---


208 Chapter 5. Administration
Capped Collections

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.

- 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 collection. 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 243) 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 createIndex() 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 210) 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.

```javascript
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:

```javascript
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 211). 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 211) 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 123) for information on creating a tailable cursor.

**Expire Data from Collections by Setting TTL**

New in version 2.2.

This document provides an introduction to MongoDB’s “time to live” or `TTL` collection feature. 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 503) 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 503), use the `db.collection.createIndex()` method with the `expireAfterSeconds` option on a field whose value is either a `date` (page 180) or an array that contains `date values` (page 180).

**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 503).

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

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

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

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

```bash
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:

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

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

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

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

*Analyzing MongoDB Performance* (page 213) Discusses some of the factors that can influence MongoDB’s performance.

*Evaluate Performance of Current Operations* (page 216) 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 216) Introduces the use of *projections* (page 62) to reduce the amount of data MongoDB sends to clients.

*Design Notes* (page 218) A collection of notes related to the architecture, design, and administration of MongoDB-based applications.

#### Analyzing MongoDB Performance

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

Some users may experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. *Locking Performance* (page 213) discusses how these can impact MongoDB’s internal locking.

Performance issues may indicate that the database is operating at capacity and that it is time to add additional capacity to the database. In particular, the application’s *working set* should fit in the available physical memory. See *Memory and the MMAPv1 Storage Engine* (page 214) for more information on the working set.

In some cases performance issues may be temporary and related to abnormal traffic load. As discussed in *Number of Connections* (page 214), scaling can help relax excessive traffic.

*Database Profiling* (page 215) can help you to understand what operations are causing degradation.

#### Locking Performance

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

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

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

`locks.deadlockCount` provide the number of times the lock acquisitions encountered deadlocks.
If \texttt{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 \texttt{globalLock.totalTime} is high relative to \texttt{uptime}, the database has existed in a lock state for a significant amount of time.

Long queries can result from ineffective use of indexes; non-optimal schema design; poor query structure; system architecture issues; or insufficient RAM resulting in \textit{page faults} (page 214) and disk reads.

**Memory and the MMAPv1 Storage Engine**

**Memory Use** With the MMAPv1 (page 91) storage engine, MongoDB uses memory-mapped files to store data. Given a data set of sufficient size, the \texttt{mongod} process will allocate all available memory on the system for its use.

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

The \textit{memory usage statuses} metrics of the \texttt{serverStatus} output can provide insight into MongoDB's memory use. The \texttt{mem.resident} field provides the amount of resident memory in use. If this exceeds the amount of system memory \textit{and} there is a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

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

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

MongoDB reports its triggered page faults as the total number of \textit{page faults} in one second. To check for page faults, see the \texttt{extra_info.page_faults} value in the \texttt{serverStatus} output.

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

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

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

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

See \textit{What are page faults?} (page 775) for more information.

**Number of Connections**

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

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

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

For read-heavy applications, increase the size of your replica set and distribute read operations to secondary members. For write-heavy applications, deploy sharding and add one or more shards to a sharded cluster to distribute load among mongod instances.

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

Unless constrained by system-wide limits, MongoDB has no limit on incoming connections. On Unix-based systems, you can modify system limits using the `ulimit` command, or by editing your system’s `/etc/sysctl` file. See **UNIX ulimit Settings** (page 287) 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:

<table>
<thead>
<tr>
<th>Level</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Off. No profiling</td>
</tr>
<tr>
<td>1</td>
<td>On. Only includes “slow” operations</td>
</tr>
<tr>
<td>2</td>
<td>On. Includes all operations</td>
</tr>
</tbody>
</table>

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

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

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

**See**

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

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

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

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

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

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

This returns all operations that lasted longer than 100 milliseconds. Ensure that the value specified here (100, in this example) is above the `slowOpThresholdMs` threshold.
You must use the $query operator to access the query field of documents within system.profile.

Additional Resources

- MongoDB Ops Optimization Consulting Package

Evaluate Performance of Current Operations

The following sections describe techniques for evaluating operational performance.

Use the Database Profiler to Evaluate Operations Against the Database

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

For more information, see Database Profiling (page 215).

Use db.currentOp() to Evaluate mongod Operations

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

Use explain to Evaluate Query Performance

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

Example

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

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

For more information, see http://docs.mongodb.org/manual/reference/explain-results, cursor.explain(), db.collection.explain(), and Analyze Query Performance (page 111).

Optimize Query Performance

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

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

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

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

**Example**

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

Creating this index:

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

Optimizes this query:

```javascript
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:

```javascript
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:

```javascript
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 107).
Use $hint to Select a Particular Index

In most cases the query optimizer (page 68) selects the optimal index for a specific operation; however, you can force MongoDB to use a specific index using the hint() method. Use hint() to support performance testing, or on some queries where you must select a field or field included in several indexes.

Use the Increment Operator to Perform Operations Server-Side

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

Design Notes

This page details features of MongoDB that may be important to 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 145) 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 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:

---

83http://bsonspec.org/#/specification
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 76).

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 150).

No Fully Generalized Transactions  MongoDB does not have fully generalized transactions (page 80). 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 559) 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 580). 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 187) for a detailed discussion of MongoDB’s monitoring options.
2. Specify appropriate write concern.
3. If your application requires manual fail over, you can configure your secondaries as priority 0 (page 567). 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 584).

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 732).

• Consider pre-splitting (page 722) an empty sharded collection before a massive bulk import.

**Analyze Performance**

As you develop and operate applications with MongoDB, you may want to analyze the performance of the database as the application. *Analyzing MongoDB Performance* (page 213) discusses some of the operational factors that can influence performance.

**Additional Resources**

• MongoDB Ops Optimization Consulting Package

**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 221) Describes routine management operations, including configuration and performance analysis.

**Manage mongod Processes** (page 224) Start, configure, and manage running mongod process.

**Rotate Log Files** (page 231) Archive the current log files and start new ones.

Continue reading from *Configuration, Maintenance, and Analysis* (page 221) for additional tutorials of fundamental MongoDB maintenance procedures.

**Backup and Recovery** (page 243) Outlines procedures for data backup and restoration with mongod instances and deployments.

**Backup and Restore with Filesystem Snapshots** (page 243) 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 254) Detailed procedures and considerations for backing up sharded clusters and single shards.

**Recover Data after an Unexpected Shutdown** (page 263) Recover data from MongoDB data files that were not properly closed or have an invalid state.

Continue reading from *Backup and Recovery* (page 243) for additional tutorials of MongoDB backup and recovery procedures.

**MongoDB Scripting** (page 265) An introduction to the scripting capabilities of the mongo shell and the scripting capabilities embedded in MongoDB instances.

**MongoDB Tutorials** (page 283) A complete list of tutorials in the MongoDB Manual that address MongoDB operation and use.

84https://www.mongodb.com/products/consulting?jmp=docs#ops_optimization
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 221) Describes Transparent Huge Pages (THP) and provides detailed instructions on disabling them.

**Use Database Commands** (page 223) The process for running database commands that provide basic database operations.

**Manage mongod Processes** (page 224) Start, configure, and manage running mongod process.

**Terminate Running Operations** (page 226) Stop in progress MongoDB client operations using `db.killOp()` and `maxTimeMS()`.

**Analyze Performance of Database Operations** (page 227) Collect data that introspects the performance of query and update operations on a mongod instance.

**Rotate Log Files** (page 231) Archive the current log files and start new ones.

**Manage Journaling** (page 233) 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 235) Describes how to store JavaScript functions on a MongoDB server.

**Upgrade to the Latest Revision of MongoDB** (page 236) Introduces the basic process for upgrading a MongoDB deployment between different minor release versions.

**Monitor MongoDB With SNMP on Linux** (page 239) The SNMP extension, available in MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

**Monitor MongoDB Windows with SNMP** (page 240) The SNMP extension, available in the Windows build of MongoDB Enterprise, allows MongoDB to provide database metrics via SNMP.

**Troubleshoot SNMP** (page 242) Outlines common errors and diagnostic processes useful for deploying MongoDB Enterprise with SNMP support.

### Disable Transparent Huge Pages (THP)

Transparent Huge Pages (THP) is a Linux memory management kernel module used to reduce the overhead of Translation Lookaside Buffer (TLB) lookups on machines with large amounts of memory, and therefore a large number of memory pages. In effect, it presents system memory as a smaller number of much larger pages rather than as a larger number of 4096-byte pages.

However, THP is known to perform poorly under database workloads, which tend to have sparse rather than contiguous memory access patterns. You must disable THP on Linux machines used to run MongoDB instances to ensure best performance.

There are three ways to disable Transparent Huge Pages on Linux.

#### In Boot-Time Configuration (Preferred)

**Step 1: Edit grub configuration file** If your system uses `grub-legacy`, you can edit the configuration file directly.

Add `transparent_hugepage=never` to the `kernel` line of your `/etc/grub.conf` or `http://docs.mongodb.org/manual/boot/grub/grub.conf` as root:
kernel <kernel path> <kernel options> transparent_hugepage=never

If your system uses grub2, you can edit the GRUB_CMDLINE_LINUX_DEFAULT value in /etc/default/grub as root to add transparent_hugepage=never to any existing kernel options:

```
GRUB_CMDLINE_LINUX_DEFAULT="<kernel options> transparent_hugepage=never"
```

and subsequently run

```
grub2-mkconfig -o <grub2 configuration file location>
```

as root.

See your operating system’s documentation for details on the precise location of the grub-legacy or grub2 configuration file.

**Step 2: Reboot the system.**

**Using tuned and ktune (Necessary if you use tuned)**

`tuned` and `ktune` are dynamic kernel tuning tools that act in accordance with policies set by the system administrator. One of the settings available to `tuned` and `ktune` is whether or not to use THP.

To disable transparent hugepages, you need to edit or create a new profile that sets THP to `never`.

**Step 1: Create a new profile**  If necessary, 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`, but you may start with any profile and name the new profile whatever you wish.

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

You may also opt to edit an existing profile.

**Step 2: Edit ktune.sh**  Locate the `ktune.sh` file in the new profile directory (`/etc/tune-profiles/no-thp/ktune.sh` in this case) 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:

```
tune-adm profile no-thp
```

where `no-thp` should be replaced with the name of the profile you defined, if it is different.

**Warning:**  If you are using `tuned` and `ktune` and do not wish to disable them, you **must** use this method, or `tuned` may override THP settings configured using either of the other two methods.
In `/etc/rc.local (Alternate)`

**Step 1: Edit `/etc/rc.local`**  
`/etc/rc.local` is a user-configurable script that is run at the end of post-boot system initialization. Add the following script lines to the file as `root` to disable THP upon each boot.

```bash
if test -f /sys/kernel/mm/transparent_hugepage/khugepaged/defrag; then
echo 0 > /sys/kernel/mm/transparent_hugepage/khugepaged/defrag
fi
if test -f /sys/kernel/mm/transparent_hugepage/defrag; then
echo never > /sys/kernel/mm/transparent_hugepage/defrag
fi
if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
  echo never > /sys/kernel/mm/transparent_hugepage/enabled
fi
```

These lines should immediately precede `exit 0`, which should already be the last line in the file. Note that on Red Hat Enterprise Linux, CentOS, and potentially other Red Hat-based derivatives, `transparent_hugepage` in the paths in the script should be replaced by `redhat_transparent_hugepages`.

**Step 2: Apply the changes to `/etc/rc.local` as root**

```
source /etc/rc.local
```

**Test Your Changes**

Whichever of the three methods you use, you can check the status of THP support by issuing the command:

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

or

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

on Red Hat Enterprise Linux, CentOS, and potentially other Red Hat-based derivatives.

Correct output resembles:

```
always madvise [never]
```

**Use Database Commands**

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

See [http://docs.mongodb.org/manual/reference/command](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:

```bash
{ 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:

```javascript
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 227) for more information on operation termination.

**admin Database Commands**

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

```javascript
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:

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

**Command Responses**

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

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

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

**Manage mongod Processes**

MongoDB runs as a standard program. You can start MongoDB from a command line by issuing the `mongod` command and specifying options. For a list of options, see the `mongod` reference. MongoDB can also run as a Windows service. For details, see *Configure a Windows Service for MongoDB* (page 30). 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 192).

**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

**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:

```javascript
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:

```javascript
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:

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

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

```javascript
{ "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.

```javascript
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**

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 228) is also configurable when enabling profiling.

The database profiler writes all the data it collects to the `system.profile` collection, which is a **capped collection** (page 209). See **Database Profiler Output** (page 292) for overview of the data in the `system.profile` (page 291) 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 292)
- 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 228) 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 228). The profiler records data in the `system.profile` (page 291) collection. MongoDB creates the `system.profile` (page 291) 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:

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

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

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

To verify the new setting, see the *Check Profiling Level* (page 229) 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:

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

**Check Profiling Level**  To view the profiling level (page 228), issue the following from the `mongo` shell:

```javascript
db.getProfilingStatus()
```

The shell returns a document similar to the following:

```json
{ "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 228).

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

```javascript
db.getProfilingLevel()
```

**Disable Profiling**  To disable profiling, use the following helper in the `mongo` shell:

```javascript
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:

```bash
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 291) collection.
To view profiling information, query the `system.profile` (page 291) 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 231).

For an explanation of the output data, see *Database Profiler Output* (page 292).

**Example Profiler Data Queries**  This section displays example queries to the `system.profile` (page 291) collection. For an explanation of the query output, see *Database Profiler Output* (page 292).

To return the most recent 10 log entries in the `system.profile` (page 291) collection, run a query similar to the following:

```javascript
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:

```javascript
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:

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

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

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

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

```javascript
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:

```javascript
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 291) 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 291) collection, you must:

1. Disable profiling.
2. Drop the `system.profile` (page 291) collection.
3. Create a new `system.profile` (page 291) collection.
4. Re-enable profiling.

For example, to create a new `system.profile` (page 291) collections that's 4000000 bytes, use the following sequence of operations in the `mongo` shell:

```javascript
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 291) 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 630).

Rotate Log Files

**Overview**

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

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

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

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

See also:

For information on logging, see the *Process Logging* (page 190) section.

5.2. Administration Tutorials 231
**Default Log Rotation Behavior**

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

**Step 1: Start a `mongod` instance.**

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

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

**Step 2: List the log files**  
In a separate terminal, list the matching files:

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

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

**Step 3: Rotate the log file.**  
Rotate the log file by issuing the `logRotate` command from the `admin` database in a `mongo` shell:

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

**Step 4: View the new log files**  
List the new log files to view the newly-created log:

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

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

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

**Log Rotation with `--logRotate reopen`**

New in version 3.0.0.

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

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

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

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

**Step 2: List the log files**  
In a separate terminal, list the matching files:

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

The results should include one log file, `server1.log`. 
Step 3: Rotate the log file. Rotate the log file by issuing the `logRotate` command from the `admin` database in a mongo shell:

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

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

Syslog Log Rotation

New in version 2.2.

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

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

```
mongod --syslog
```

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

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

Step 2: Rotate the log. Store and rotate the log output using your system’s default log rotation mechanism.

Forcing a Log Rotation with SIGUSR1

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

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

Manage Journaling

MongoDB uses write ahead logging to an on-disk journal to guarantee write operation (page 73) 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 263) or, preferably, resync (page 633) 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 309).

Important: Changed in version 2.0: For 64-bit builds of `mongod`, journaling is enabled by default. For other platforms, see `storage.journal.enabled`. 

5.2. Administration Tutorials 233
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.

Warning: Do not disable journaling on production systems. If your mongod instance stops without shutting down cleanly unexpectedly for any reason, (e.g. power failure) and you are not running with journaling, then you must recover from an unaffected replica set member or backup, as described in repair (page 263).

Disable Journaling

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

Get Commit Acknowledgment  You can get commit acknowledgment with the Write Concern (page 78) and the j option. For details, see Write Concern Reference (page 130).

Avoid Preallocation Lag  To avoid preallocation lag (page 308), you can preallocate files in the journal directory by copying them from another instance of mongod.

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

Example

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

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

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

   mkdir ~/tmpDbpath

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

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

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

   [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 examples:

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

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

- The `_id` field holds the name of the function and is unique per database.
- The `value` field holds the function definition.
Once you save a function in the `system.js` collection, you can use the function from any JavaScript context; e.g. `$where` operator, `mapReduce` command or `db.collection.mapReduce()`.

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

```javascript
db.loadServerScripts();

echoFunction(3);

myAddFunction(3, 5);

Upgrade to the Latest Revision of MongoDB

Revisions provide security patches, bug fixes, and new or changed features that do not contain any backward breaking changes. Always upgrade to the latest revision in your release series. The third number in the MongoDB version number (page 924) indicates the revision.

Before Upgrading

- Ensure you have an up-to-date backup of your data set. See MongoDB Backup Methods (page 184).
- Consult the following documents for any special considerations or compatibility issues specific to your MongoDB release:
  - The release notes, located at Release Notes (page 785).
  - The documentation for your driver. See Drivers 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 237).

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 237).
3. Upgrade any standalone instances. See Upgrade a MongoDB Instance (page 237).
4. Upgrade any replica sets that are not part of a sharded cluster, as described in Upgrade Replica Sets (page 238).

85https://docs.mongodb.org/ecosystem/drivers
Upgrade a MongoDB Instance

To upgrade a \texttt{mongod} or \texttt{mongos} instance, use one of the following approaches:

- Upgrade the instance using the operating system’s package management tool and the official MongoDB packages. This is the preferred approach. See \textit{Install MongoDB} (page 5).
- Upgrade the instance by replacing the existing binaries with new binaries. See \textit{Replace the Existing Binaries} (page 237).

Replace the Existing Binaries

\textbf{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 \textit{Install MongoDB} (page 5).

To upgrade a \texttt{mongod} or \texttt{mongos} instance by replacing the existing binaries:

1. Download the binaries for the latest MongoDB revision from the MongoDB Download Page\footnote{http://downloads.mongodb.org/} 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 \textit{Disable the Balancer} (page 717).
2. Upgrade each \texttt{mongos} instance by following the instructions below in \textit{Upgrade a MongoDB Instance} (page 237). You can upgrade the \texttt{mongos} instances in any order.
3. Upgrade each \texttt{mongod config server} (page 670) individually starting with the last config server listed in your \texttt{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 \textit{Upgrade a MongoDB Instance} (page 237)

\textbf{Example}

Given the following config string:

\texttt{mongos --configdb \textit{cfg0.example.net:27019,cfg1.example.net:27019,cfg2.example.net:27019}}

You would upgrade the config servers in the following order:

(a) \texttt{cfg2.example.net}
(b) \texttt{cfg1.example.net}
(c) \texttt{cfg0.example.net}
Upgrade each shard.

- If a shard is a replica set, upgrade the shard using the procedure below titled *Upgrade Replica Sets* (page 238).
- If a shard is a standalone instance, upgrade the shard using the procedure below titled *Upgrade a MongoDB Instance* (page 237).

5. Re-enable the balancer, as described in *Enable the Balancer* (page 718).

### 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 237).
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 580) 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 237).

### Additional Resources

- *MongoDB Major Version Upgrade Consulting Package*[^87]

[^87]: https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade
Monitor MongoDB With SNMP on Linux

New in version 2.2.

Enterprise Feature
SNMP is only available in MongoDB Enterprise88.

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 242) and Monitor MongoDB Windows with SNMP (page 240) for complete instructions on using MongoDB with SNMP on Windows systems.

Considerations

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

Configuration Files

Changed in version 2.6.

MongoDB Enterprise contains the following configuration files to support SNMP:

- MONGOD-MIB.txt:
  The management information base (MIB) file that defines MongoDB’s SNMP output.
- mongod.conf.subagent:
  The configuration file to run mongod as the SNMP subagent. This file sets SNMP run-time configuration options, including the AgentX socket to connect to the SNMP master.
- mongod.conf.master:
  The configuration file to run mongod as the SNMP master. This file sets SNMP run-time configuration options.

Procedure

Step 1: Copy configuration files. Use the following sequence of commands to move the SNMP configuration files to the SNMP service configuration directory.

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

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

88http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
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 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**

New in version 2.6.

**Enterprise Feature**

SNMP is only available in MongoDB Enterprise.

**Overview**

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This 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 239) and *Troubleshoot SNMP* (page 242) 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:

```bash
md C:\snmp\etc\config
copy MONGOD-MIB.txt C:\snmp\etc\config\MONGOD-MIB.txt
copy mongod.conf.subagent C:\snmp\etc\config\mongod.conf
```

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

Edit the configuration file to ensure that the communication between the agent (i.e. `snmpd` or the master) and sub-agent (i.e. MongoDB) uses TCP.

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

**Step 2: Start MongoDB.** Start `mongod.exe` with the `snmp-subagent` to send data to the SNMP master.

`mongod.exe --snmp-subagent`

**Step 3: Confirm SNMP data retrieval.** Use `snmpwalk` to collect data from `mongod.exe`.

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

Install the net-snmp\(^9\) package to access the `snmpwalk` client. `net-snmp` provides the `snmpwalk` SNMP client.

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

New in version 2.6.

Enterprise Feature

SNMP is only available in MongoDB Enterprise.

Overview

MongoDB Enterprise can provide database metrics via SNMP, in support of centralized data collection and aggregation. This document identifies common problems you may encounter when deploying MongoDB Enterprise with SNMP as well as possible solutions for these issues.

See Monitor MongoDB With SNMP on Linux (page 239) and Monitor MongoDB Windows with SNMP (page 240) 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 274192. 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.

92http://www.ietf.org/rfc/rfc2741.txt
Error Parsing Command Line  One of the following errors at the command line:

```bash
Error parsing command line: unknown option snmp-master
try 'mongod --help' for more information
```
```bash
Error parsing command line: unknown option snmp-subagent
try 'mongod --help' for more information
```
mongodb binaries that are not part of the Enterprise Edition produce this error. Install the Enterprise Edition (page 33) and attempt to start mongod again.

Other MongoDB binaries, including mongos will produce this error if you attempt to start 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.

### 5.2.2 Backup and Recovery

The following tutorials describe backup and restoration for a mongod instance:

- **Backup and Restore with Filesystem Snapshots** (page 243)  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 247)  Describes procedure for restoring a replica set from an archived backup such as a mongodump or MongoDB Cloud Manager\(^9\) Backup file.
- **Back Up and Restore with MongoDB Tools** (page 249)  Describes a procedure for exporting the contents of a database to either a binary dump or a textual exchange format, and for importing these files into a database.
- **Backup and Restore Sharded Clusters** (page 254)  Detailed procedures and considerations for backing up sharded clusters and single shards.
- **Recover Data after an Unexpected Shutdown** (page 263)  Recover data from MongoDB data files that were not properly closed or have an invalid state.

**Backup and Restore with Filesystem Snapshots**

This document describes a procedure for creating backups of MongoDB systems using system-level tools, such as LVM or storage appliance, as well as the corresponding restoration strategies.

These filesystem snapshots, or “block-level” backup methods use system level tools to create copies of the device that holds MongoDB’s data files. These methods complete quickly and work reliably, but require more system configuration outside of MongoDB.

See also:

- **MongoDB Backup Methods** (page 184) and **Back Up and Restore with MongoDB Tools** (page 249).

\(^9\)[https://cloud.mongodb.com/?jmp=docs](https://cloud.mongodb.com/?jmp=docs)
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 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 246).

- 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 245).
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 the 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 246) 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: This procedure is only supported with the MMAPv1 storage engine.

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

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 243). The MongoDB Cloud Manager produces MongoDB database files for stored snapshots and point in time snapshots. For an Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced, see also documentation on Backup.

You can also use mongorestore to restore database files using data created with mongodump. See Back Up and Restore with MongoDB Tools (page 249) for more information.

Step 2: Start a mongod using data files from the backup as the data path. The following example uses /data/db as the data path, as specified in the dbpath setting:

mongod --dbpath /data/db

5.2. Administration Tutorials 247
Step 3: Convert the standalone `mongod` to a single-node replica set. Convert the standalone `mongod` process to a single-node replica set by shutting down the `mongod` instance, and restarting it with the `--replSet` option, as in the following example:

```
mongod --dbpath /data/db --replSet <replName>
```

Optionally, you can explicitly set a `oplogSizeMB` to control the size of the `oplog` created for this replica set member.

Step 4: Connect to the `mongod` instance. For example, first use the following command to a `mongod` instance running on the localhost interface:

```
mongo
```

Step 5: Initiate the new replica set. Use `rs.initiate()` to initiate the new replica set, as in the following example:

```
rs.initiate()
```

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 594) 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 603) 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 594) copies the data from the primary to the new member.

Back Up and Restore with MongoDB Tools

This document describes the process for creating backups and restoring data using the utilities provided with MongoDB.

Because all of these tools primarily operate by interacting with a running mongod instance, they can impact the performance of your running database.

Not only do they create traffic for a running database instance, they also force the database to read all data through memory. When MongoDB reads infrequently used data, it can supplant more frequently accessed data, causing a deterioration in performance for the database’s regular workload.

No matter how you decide to import or export your data, consider the following guidelines:

- Label files so that you can identify the contents of the export or backup as well as the point in time the export/backup reflects.
- Do not create or apply exports if the backup process itself will have an adverse effect on a production system.
- Make sure that the backups reflect a consistent data state. Export or backup processes can impact data integrity (i.e. type fidelity) and consistency if updates continue during the backup process.
- Test backups and exports by restoring and importing to ensure that the backups are useful.

See also:

MongoDB Backup Methods (page 184) or MongoDB Cloud Manager Backup documentation for more information on backing up MongoDB instances. Additionally, consider the following references for the MongoDB import/export tools:

- mongoimport
- mongoexport
- mongorestore
- mongodump

Binary BSON Dumps

The mongorestore and mongodump utilities work with BSON (page 178) data dumps, and are useful for creating backups of small deployments. For resilient and non-disruptive backups, use a file system or block-level disk snapshot function, such as the methods described in the MongoDB Backup Methods (page 184) document.

Use these tools for backups if other backup methods, such as the MongoDB Cloud Manager or file system snapshots (page 243) are unavailable.

99 https://docs.cloud.mongodb.com/tutorial/nav/backup-use/
100 https://cloud.mongodb.com/?jmp=docs
Backup a Database with `mongodump`  
mongodump does not dump the content of the local database.

To back up all the databases in a cluster via `mongodump`, you should have the backup (page 410) role. The backup (page 410) role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of least privilege.

To back up a given database, you must have read access on the database. Several roles provide this access, including the backup (page 410) role.

To back up the `system.profile` (page 291) collection, which is created when you activate `database profiling` (page 215), you must have additional read access on this collection. Several roles provide this access, including the `clusterAdmin` (page 408) and `dbAdmin` (page 406) roles.

Changed in version 2.6.

To back up users and `user-defined roles` (page 325) for a given database, you must have access to the `admin` database. MongoDB stores the user data and role definitions for all databases in the `admin` database.

Specifically, to back up a given database’s users, you must have the `find` (page 419) `action` (page 419) on the `admin` database’s `admin.system.users` (page 291) collection. The backup (page 410) and `userAdminAnyDatabase` (page 412) roles both provide this privilege.

To back up the user-defined roles on a database, you must have the `find` (page 419) action on the `admin` database’s `admin.system.roles` (page 291) collection. Both the backup (page 410) and `userAdminAnyDatabase` (page 412) roles provide this privilege.

**Basic mongodump Operations**  
The `mongodump` utility backs up data by connecting to a running `mongod` or `mongos` instance.

The utility can create a backup for an entire server, database or collection, or can use a query to backup just part of a collection.

When you run `mongodump` without any arguments, the command connects to the MongoDB instance on the local system (e.g. 127.0.0.1 or localhost) on port 27017 and creates a database backup named `dump/` in the current directory.

To backup data from a `mongod` or `mongos` instance running on the same machine and on the default port of 27017, use the following command:

```
mongodump
```

The data format used by `mongodump` from version 2.2 or later is incompatible with earlier versions of `mongod`. Do not use recent versions of `mongodump` to back up older data stores.

You can also specify the `--host` and `--port` of the MongoDB instance that the `mongodump` should connect to. For example:

```
mongodump --host mongodb.example.net --port 27017
```

`mongodump` will write BSON files that hold a copy of data accessible via the `mongod` listening on port 27017 of the `mongodb.example.net` host. See Create Backups from Non-Local mongod Instances (page 251) for more information.

To specify a different output directory, you can use the `--out` or `-o` option:

```
mongodump --out /data/backup/
```

To limit the amount of data included in the database dump, you can specify `--db` and `--collection` as options to `mongodump`. For example:
mongodump --collection myCollection --db test

This operation creates a dump of the collection named myCollection from the database test in a dump/subdirectory of the current working directory.

mongodump overwrites output files if they exist in the backup data folder. Before running the mongodump command multiple times, either ensure that you no longer need the files in the output folder (the default is the dump/ folder) or rename the folders or files.

Point in Time Operation Using Oplogs Use the --oplog option with mongodump to collect the oplog entries to build a point-in-time snapshot of a database within a replica set. With --oplog, mongodump copies all the data from the source database as well as all of the oplog entries from the beginning to the end of the backup procedure. This operation, in conjunction with mongorestore --oplogReplay, allows you to restore a backup that reflects the specific moment in time that corresponds to when mongodump completed creating the dump file.

Create Backups from Non-Local mongod Instances The --host and --port options for mongodump allow you to connect to and backup from a remote host. Consider the following example:

mongodump --host mongodb1.example.net --port 3017 --username user --password pass --out /opt/backup/mongodump-2013-10-24

On any mongodump command you may, as above, specify username and password credentials to specify database authentication.

Restore a Database with mongorestore To restore collection data to a database with authentication enabled, the connecting user must possess the appropriate user roles.

To restore a single database, the connecting user must possess the readWrite (page 406) role for that database. Alternatively, the readWriteAnyDatabase (page 412) provides access to restore any database. The restore (page 411) role also provides the requisite permissions.

Changed in version 2.6.

To restore users and user-defined roles (page 325) 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 419) on the admin database’s admin.system.users (page 291) collection. The restore (page 411) 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 291) collection. The restore (page 411) role provides this privilege.

If your database is running with authentication enabled, you must possess the userAdmin (page 407) role on the database you are restoring, or the userAdminAnyDatabase (page 412) role, which allows you to restore user data to any database. The restore (page 411) role also provides the requisite privileges.

Basic mongorestore Operations The mongorestore utility restores a binary backup created by mongodump. By default, mongorestore looks for a database backup in the dump/ directory.

The mongorestore utility restores data by connecting to a running mongod or mongos directly.

mongorestore can restore either an entire database backup or a subset of the backup.

To use mongorestore to connect to an active mongod or mongos, use a command with the following prototype form:
mongorestore --port <port number> <path to the backup>

Consider the following example:

mongorestore dump-2013-10-25/

Here, mongorestore imports the database backup in the dump-2013-10-25 directory to the mongod instance running on the localhost interface.

**Restore Point in Time Oplog Backup** If you created your database dump using the `--oplog` option to ensure a point-in-time snapshot, call mongorestore with the `--oplogReplay` option, as in the following example:

`mongorestore --oplogReplay`

You may also consider using the `mongorestore --objcheck` option to check the integrity of objects while inserting them into the database, or you may consider the `mongorestore --drop` option to drop each collection from the database before restoring from backups.

**Restore Backups to Non-Local mongod Instances** By default, mongorestore connects to a MongoDB instance running on the localhost interface (e.g. 127.0.0.1) and on the default port (27017). If you want to restore to a different host or port, use the `--host` and `--port` options.

Consider the following example:

`mongorestore --host mongodb1.example.net --port 3017 --username user --password pass /opt/backup/mongodump-2013-10-24`

As above, you may specify username and password connections if your mongod requires authentication.

**Human Intelligible Import/Export Formats**

MongoDB’s `mongoimport` and `mongoexport` tools allow you to work with your data in a human-readable Extended JSON or CSV format. This is useful for simple ingestion to or from a third-party system, and when you want to backup or export a small subset of your data. For more complex data migration tasks, you may want to write your own import and export scripts using a client driver to interact with the database.

The examples in this section use the MongoDB tools `mongoimport` and `mongoexport`. These tools may also be useful for importing data into a MongoDB database from third party applications.

If you want to simply copy a database or collection from one instance to another, consider using the `copydb`, `clone`, or `cloneCollection` commands, which may be more suited to this task. The mongo shell provides the `db.copyDatabase()` method.

**Warning:** Avoid using `mongoimport` and `mongoexport` for full instance production backups. They do not reliably preserve all rich BSON data types, because `JSON` can only represent a subset of the types supported by BSON. Use `mongodump` and `mongorestore` as described in *MongoDB Backup Methods* (page 184) for this kind of functionality.

**Collection Export with mongoexport**
Export in CSV Format  Changed in version 3.0.0: mongoexport removed the --csv option. Use the --type=csv option to specify CSV format for the output.

In the following example, mongoexport exports data from the collection contacts collection in the users database in CSV format to the file /opt/backups/contacts.csv.

The mongod instance that mongoexport connects to is running on the localhost port number 27017.

When you export in CSV format, you must specify the fields in the documents to export. The operation specifies the name and address fields to export.

```
mongoexport --db users --collection contacts --type=csv --fields name,address --out /opt/backups/contacts.csv
```

For CSV exports only, you can also specify the fields in a file containing the line-separated list of fields to export. The file must have only one field per line.

For example, you can specify the name and address fields in a file fields.txt:

```
name
address
```

Then, using the --fieldFile option, specify the fields to export with the file:

```
mongoexport --db users --collection contacts --type=csv --fieldFile fields.txt --out /opt/backups/contacts.csv
```

Export in JSON Format  This example creates an export of the contacts collection from the MongoDB instance running on the localhost port number 27017, with journaling explicitly enabled. This writes the export to the contacts.json file in JSON format.

```
mongoexport --db sales --collection contacts --out contacts.json --journal
```

Export from Remote Host Running with Authentication  The following example exports the contacts collection from the MongoDB instance running on the localhost port number 27017, with journaling explicitly enabled. This writes the export to the contacts.json file in JSON format.

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection contacts --out contacts.json
```

Export Query Results  You can export only the results of a query by supplying a query filter with the --query option, and limit the results to a single database using the "--db" option.

For instance, this command returns all documents in the sales database's contacts collection that contain a field named field with a value of 1.

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

You must enclose the query in single quotes (e.g. ') to ensure that it does not interact with your shell environment.

Collection Import with mongoimport
Simple Usage  
mongoimport restores a database from a backup taken with mongoexport. Most of the arguments to mongoexport also exist for mongoimport.

In the following example, mongoimport imports the data in the JSON data from the contacts.json file into the collection contacts in the users database.

```
mongoimport --db users --collection contacts --file contacts.json
```

Import JSON to Remote Host Running with Authentication  
In the following example, mongoimport imports data from the file `/opt/backups/mdb1-examplenet.json` into the contacts collection within the database marketing on a remote MongoDB database with authentication enabled.

```
mongoimport connects to the mongod instance running on the host mongodb1.example.net over port 37017. It authenticates with the username user and the password pass.
```

```
mongoimport --host mongodb1.example.net --port 37017 --username user --password pass --collection contacts --file /opt/backups/mdb1-examplenet.json
```

CSV Import  
In the following example, mongoimport imports the csv formatted data in the `/opt/backups/contacts.csv` file into the collection contacts in the users database on the MongoDB instance running on the localhost port numbered 27017.

Specifying `--headerline` instructs mongoimport to determine the name of the fields using the first line in the CSV file.

```
mongoimport --db users --collection contacts --type csv --headerline --file /opt/backups/contacts.csv
```

mongoimport uses the input file name, without the extension, as the collection name if `-c` or `--collection` is unspecified. The following example is therefore equivalent:

```
mongoimport --db users --type csv --headerline --file /opt/backups/contacts.csv
```

Use the `"--ignoreBlanks"` option to ignore blank fields. For CSV and TSV imports, this option provides the desired functionality in most cases because it avoids inserting fields with null values into your collection.

Additional Resources

- Backup and its Role in Disaster Recovery White Paper[^101]
- Cloud Backup through MongoDB Cloud Manager[^102]
- Blog Post: Backup vs. Replication, Why you Need Both[^103]
- Backup Service with Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced[^104]

Backup and Restore Sharded Clusters

The following tutorials describe backup and restoration for sharded clusters:

**Backup a Small Sharded Cluster with mongodump (page 255)**  
If your sharded cluster holds a small data set, you can use mongodump to capture the entire backup in a reasonable amount of time.

[^102]: https://cloud.mongodb.com/?jmp=docs
[^104]: https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
Backup a Sharded Cluster with Filesystem Snapshots (page 256) Use file system snapshots back up each component in the sharded cluster individually. The procedure involves stopping the cluster balancer. If your system configuration allows file system backups, this might be more efficient than using MongoDB tools.

Backup a Sharded Cluster with Database Dumps (page 258) Create backups using mongodump to back up each component in the cluster individually.

Schedule Backup Window for Sharded Clusters (page 259) Limit the operation of the cluster balancer to provide a window for regular backup operations.

Restore a Single Shard (page 260) An outline of the procedure and consideration for restoring a single shard from a backup.

Restore a Sharded Cluster (page 260) An outline of the procedure and consideration for restoring an entire sharded cluster from backup.

Backup a Small Sharded Cluster with mongodump

Overview If your sharded cluster holds a small data set, you can connect to a mongos using mongodump. You can create backups of your MongoDB cluster, if your backup infrastructure can capture the entire backup in a reasonable amount of time and if you have a storage system that can hold the complete MongoDB data set.

See MongoDB Backup Methods (page 184) and Backup and Restore Sharded Clusters (page 254) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Important: By default mongodump issue its queries to the non-primary nodes.

To back up all the databases in a cluster via mongodump, you should have the backup (page 410) role. The backup (page 410) role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of least privilege.

To back up a given database, you must have read access on the database. Several roles provide this access, including the backup (page 410) role.

To back up the system.profile (page 291) collection, which is created when you activate database profiling (page 215), you must have additional read access on this collection. Several roles provide this access, including the clusterAdmin (page 408) and dbAdmin (page 406) roles.

Changed in version 2.6.

To back up users and user-defined roles (page 325) for a given database, you must have access to the admin database. MongoDB stores the user data and role definitions for all databases in the admin database.

Specifically, to back up a given database’s users, you must have the find (page 419) action (page 419) on the admin database’s admin.system.users (page 291) collection. The backup (page 410) and userAdminAnyDatabase (page 412) roles both provide this privilege.

To back up the user-defined roles on a database, you must have the find (page 419) action on the admin database’s admin.system.roles (page 291) collection. Both the backup (page 410) and userAdminAnyDatabase (page 412) 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 670).

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 691) for more information.

Backup a Sharded Cluster with Filesystem Snapshots

Overview This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses file system snapshots to capture a copy of the mongod instance. An alternate procedure uses mongodump to create binary database dumps when file-system snapshots are not available. See Backup a Sharded Cluster with Database Dumps (page 258) for the alternate procedure.

See MongoDB Backup Methods (page 184) and Backup and Restore Sharded Clusters (page 254) 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 \texttt{sh.stopBalancer()} method in the \texttt{mongo} shell.

Consider the following example:

\begin{verbatim}
use config
sh.stopBalancer()
\end{verbatim}

For more information, see the \textit{Disable the Balancer} (page 717) 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 backup.

If your mongod has journaling enabled and your journal and data files are on the same volume, you may skip this step.

If you need to lock the mongod, attempt to lock one secondary member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time.

To lock a secondary, connect through the \texttt{mongo} shell to the secondary member’s mongod instance and issue the \texttt{db.fsyncLock()} method.

**Step 3: Back up one of the config servers.** Backing up a config server (page 670) 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 \textit{Backup and Restore with Filesystem Snapshots} (page 243). Never use \texttt{db.fsyncLock()} on config databases.

- Create a database dump to backup the config server. Issue \texttt{mongodump} against one of the config mongod instances. If you are running MongoDB 2.4 or later with the \texttt{--configsvr} option, then include the \texttt{--oplog} option to ensure that the dump includes a partial oplog containing operations from the duration of the mongodump operation. For example:

  \begin{verbatim}
mongodump --oplog
  \end{verbatim}

**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 \textit{Backup and Restore with Filesystem Snapshots} (page 243).

**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 \texttt{db.fsyncUnlock()} method in the \texttt{mongo} shell.

**Step 6: Enable the balancer.** Re-enable the balancer with the \texttt{sh.setBalancerState()} method. Use the following command sequence when connected to the mongos with the \texttt{mongo} shell:
use config
sh.setBalancerState(true)

Backup a Sharded Cluster with Database Dumps

Overview This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses mongodump to create dumps of the mongod instance. An alternate procedure uses file system snapshots to capture the backup data, and may be more efficient in some situations if your system configuration allows file system backups. See Backup and Restore Sharded Clusters (page 254) for more information.

See MongoDB Backup Methods (page 184) and Backup and Restore Sharded Clusters (page 254) for complete information on backups in MongoDB and backups of sharded clusters in particular.

Prerequisites Important: To capture a point-in-time backup from a sharded cluster you must stop all writes to the cluster. On a running production system, you can only capture an approximation of point-in-time snapshot.

To back up all the databases in a cluster via mongodump, you should have the backup (page 410) role. The backup (page 410) role provides the required privileges for backing up all databases. The role confers no additional access, in keeping with the policy of least privilege.

To back up a given database, you must have read access on the database. Several roles provide this access, including the backup (page 410) role.

To back up the system.profile (page 291) collection, which is created when you activate database profiling (page 215), you must have additional read access on this collection. Several roles provide this access, including the clusterAdmin (page 408) and dbAdmin (page 406) roles.

Changed in version 2.6.

To back up users and user-defined roles (page 325) for a given database, you must have access to the admin database. MongoDB stores the user data and role definitions for all databases in the admin database.

Specifically, to back up a given database’s users, you must have the find (page 419) action (page 419) on the admin database’s admin.system.users (page 291) collection. The backup (page 410) and userAdminAnyDatabase (page 412) roles both provide this privilege.

To back up the user-defined roles on a database, you must have the find (page 419) action on the admin database’s admin.system.roles (page 291) collection. Both the backup (page 410) and userAdminAnyDatabase (page 412) roles provide this privilege.

Consideration To create these backups of a sharded cluster, you will stop the cluster balancer and take a backup of the config database, and then take backups of each shard in the cluster using mongodump to capture the backup data. To capture a more exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, taking the backup from a single offline secondary member of the replica set that provides each shard can improve the quality of the backup while minimizing impact on the cluster.

Procedure

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 717) procedure.

**Warning:** If you do not stop the balancer, the backup could have duplicate data or omit data as chunks migrate while recording backups.

**Step 2: Lock replica set members.** Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these mongod instances in as short of an interval as possible.

To lock or freeze a sharded cluster, issue `db.fsyncLock()` on a member of each replica set in the cluster. Ensure that the oplog has sufficient capacity to allow these secondaries to catch up to the state of the primaries after finishing the backup procedure. See Oplog Size (page 593) 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 670).

```
mongodump --oplog
```

**Step 4: Backup replica set members.** Back up the “frozen” replica set members of the shards using `mongodump` and specifying the `--oplog` option. You may back up the shards in parallel. Consider the following invocation:

```
mongodump --oplog --out /data/backup/
```

You must run `mongodump` on the same system where the `mongod` ran. `mongodump` writes the output of this dump as well as the `oplog.bson` file to the `/data/backup/` directory.

**Step 5: Unlock replica set members.** Use `db.fsyncUnlock()` to unlock the locked replica set members of each shard. Allow these members to catch up with the state of the primary.

**Step 6: Re-enable the balancer process.** Re-enable the balancer with the `sh.setBalancerState()` method.

Use the following command sequence when connected to the mongos with the mongo shell:

```
use config
sh.setBalancerState(true)
```

**Schedule Backup Window for Sharded Clusters**

**Overview** In a sharded cluster, the balancer process is responsible for distributing sharded data around the cluster, so that each shard has roughly the same amount of data.

However, when creating backups from a sharded cluster it is important that you disable the balancer while taking backups to ensure that no chunk migrations affect the content of the backup captured by the backup procedure. Using the procedure outlined in the section Disable the Balancer (page 717) 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:

```plaintext
use config
db.settings.update({ _id: "balancer" }, { $set: { activeWindow: { start: "6:00", stop: "23:00" } } })
```

This operation configures the balancer to run between 6:00am and 11:00pm, server time. Schedule your backup operation to run and complete outside of this time. Ensure that the backup can complete outside the window when the balancer is running and that the balancer can effectively balance the collection among the shards in the window allotted to each.

**Restore a Single Shard**

**Overview** Restoring a single shard from backup with other unaffected shards requires a number of special considerations and practices. This document outlines the additional tasks you must perform when restoring a single shard.

Consider the following resources on backups in general as well as backup and restoration of sharded clusters specifically:

- [Backup and Restore Sharded Clusters](page 254)
- [Restore a Sharded Cluster](page 260)
- [MongoDB Backup Methods](page 184)

**Procedure** Always restore the single shard 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 184) 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 737).

**Restore a Sharded Cluster**

**Overview** You can restore a sharded cluster either from snapshots (page 243) or from BSON database dumps (page 258) created by the mongodump tool. This document describes procedures to

- [Restore a Sharded Cluster with Filesystem Snapshots](page 261)
- [Restore a Sharded Cluster with Database Dumps](page 262)

**Procedures**
Restore a Sharded Cluster with Filesystem Snapshots   The following procedure outlines the steps to restore a sharded cluster from filesystem snapshots. For information on using filesystem snapshots to backup sharded clusters, see Backup a Sharded Cluster with Filesystem Snapshots (page 256).

Step 1: Shut down the entire cluster.   Stop all mongos and mongod processes, including all shards and all config servers. To stop all members, connect to each member and issue following operations:

use admin
db.shutdownServer()

For version 2.4 or earlier, use db.shutdownServer({force:true}).

Step 2: Restore the data files.   On each server, extract the data files to the location where the mongod instance will access them and restore the following:

- Data files for each server in each shard.
  Because each production shard is a replica set, for each shard, restore all the members of the replica set. See Restore a Replica Set from MongoDB Backups (page 247).
- Data files for each config server.

See also: Restore a Snapshot (page 246)

Step 3: Restart the config servers.   Restart each config server mongod instance by issuing a command similar to the following for each, using values appropriate to your configuration:

mongod --configsvr --dbpath /data/configdb --port 27019

Step 4: Start one mongos instance.   Start one mongos instance. For the --configdb, specify the hostnames (and port numbers) of the config servers started in the step Restart the config servers. (page ??)

Step 5: If shard hostnames have changed, update the config database.   If shard hostnames have changed, update the shards (page 742) collection in the Config Database (page 737) to reflect the new hostnames.

Step 6: Restart all the shard mongod instances.

Step 7: Restart the other mongos instances.   Restart the remaining two mongos instances. For the --configdb, specify the hostnames (and port numbers) of the config servers started in the step Restart the config servers. (page ??)

Step 8: Verify that the cluster is operational.   Connect to a mongos instance from a mongo shell and use the db.printShardingStatus() method to ensure that the cluster is operational, as follows:

db.printShardingStatus()
show collections
Restore a Sharded Cluster with Database Dumps  

Changed in version 3.0: mongorestore requires a running MongoDB instances. Earlier versions of mongorestore did not require a running MongoDB instances and instead used the --dbpath option. For instructions specific to your version of mongorestore, refer to the appropriate version of the manual.

The following procedure outlines the steps to restore a sharded cluster from the BSON database dumps created by mongodump. For information on using mongodump to backup sharded clusters, see Backup a Sharded Cluster with Database Dumps (page 258)

The procedure deploys a new sharded cluster and restores data from database dumps.

**Step 1: Deploy a new replica set for each shard.**  For each shard, deploy a new replica set:

1. Start a new mongod for each member of the replica set. Include any other configuration as appropriate.
2. Connect a mongo to one of the mongod instances. In the mongo shell:
   
   (a) Run rs.initiate().
   
   (b) Use rs.add() to add the other members of the replica set.

For detailed instructions on deploying a replica set, see Deploy a Replica Set (page 603).

**Step 2: Deploy three new config servers.**  Start three mongod instances for the config servers (i.e. mongod --configsvr). Include any other configuration as appropriate.

For detailed instructions on setting up the config servers, see Start the Config Server Database Instances (page 691).

**Step 3: Start the mongos instances.**  Start the mongos instances, specifying the new config servers with --configdb. Include any other configuration as appropriate.

For detailed instructions on starting the mongos instances for a sharded cluster, see Start the mongos Instances (page 692).

**Step 4: Add shards to the cluster.**  Connect a mongo shell to a mongos instance. Use sh.addShard() to add each replica sets as a shard.

For detailed instructions in adding shards to the cluster, see Add Shards to the Cluster (page 692).

**Step 5: Shut down the mongos instances.**  Once the new sharded cluster is up, shut down all mongos instances.

**Step 6: Restore the shard data.**  For each shard, use mongorestore to restore the data dump to the primary’s data directory. Include the --drop option to drop the collections before restoring and, because the backup procedure (page 258) included the --oplog option, include the --oplogReplay option for mongorestore.

For example, on the primary for ShardA, run the mongorestore. Specify any other configuration as appropriate.

```
mongorestore --drop --oplogReplay /data/dump/shardA
```

After you have finished restoring all the shards, shut down all shard instances.
**Step 7: Restore the config server data.** For each config server, use `mongorestore` to restore the data dump to each config server’s data directory. Include the `--drop` option to drop the collections before restoring and, because the backup procedure (page 258) included the `--oplog` option, include the `--oplogReplay` option for mongorestore.

```
mongorestore --drop --oplogReplay /data/dump/configData
```

**Step 8: Start one mongos instance.** Start one mongos instance. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Deploy three new config servers. (page ??)

**Step 9: If shard hostnames have changed, update the config database.** If shard hostnames have changed, update the `shards` (page 742) collection in the `Config Database` (page 737) to reflect the new hostnames.

**Step 10: Restart all the shard mongod instances.**

**Step 11: Restart the other mongos instances.** Restart the remaining two mongos instances. For the `--configdb`, specify the hostnames (and port numbers) of the config servers started in the step Deploy three new config servers. (page ??)

**Step 12: Verify that the cluster is operational.** Connect to a mongos instance from a mongo shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

```
db.printShardingStatus()
show collections
```

See also:

*MongoDB Backup Methods* (page 184), *Backup and Restore Sharded Clusters* (page 254)

**Recover Data after an Unexpected Shutdown**

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

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 183) documents, including *Replica Set Syncing* (page 593), and the documentation on the `--repair repairPath` and `storage.journal.enabled` settings.

---

105 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)`.

106 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.
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 593) 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 output:

Unclean shutdown detected.

This indicates that you need to run mongod with the --repair option. If you run repair when the mongod.lock file exists in your dbPath, or the optional --repairpath, you will see a message that contains the following line:

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

If you see this message, as a last resort you may remove the lockfile and run the repair operation before starting the database normally, as in the following procedure:

Overview

Warning: Recovering a member of a replica set.
Do not use this procedure to recover a member of a replica set. Instead you should either restore from a backup (page 184) 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 633).

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

- Use the --repair option in conjunction with the --repairpath option. mongod will read the existing data files, and write the existing data to new data files.
  You do not need to remove the mongod.lock file before using this procedure.
- Use the --repair option. mongod will read the existing data files, write the existing data to new files and replace the existing, possibly corrupt, files with new files.
  You must remove the mongod.lock file before using this procedure.

Note: --repair functionality is also available in the shell with the db.repairDatabase() helper for the repairDatabase command.

Procedures

Important: Always Run mongod as the same user to avoid changing the permissions of the MongoDB data files.

Repair Data Files and Preserve Original Files  To repair your data files using the --repairpath option to preserve the original data files unmodified.

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:

```bash
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:

```bash
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:

```bash
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:

```bash
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:

```bash
mongod --dbpath /data/db
```

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 184) 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 633).

5.2.3 MongoDB Scripting

The mongo shell is an interactive JavaScript shell for MongoDB, and is part of all MongoDB distributions. This section provides an introduction to the shell, and outlines key functions, operations, and use of the mongo shell. Also consider FAQ: The mongo Shell (page 756) and the shell method and other relevant reference material.

---

107 http://www.mongodb.org/downloads
Note: Most examples in the MongoDB Manual use the mongo shell; however, many drivers provide similar interfaces to MongoDB.

**Server-side JavaScript (page 266)** Details MongoDB’s support for executing JavaScript code for server-side operations.

**Data Types in the mongo Shell (page 267)** Describes the super-set of JSON available for use in the mongo shell.

**Write Scripts for the mongo Shell (page 270)** An introduction to the mongo shell for writing scripts to manipulate data and administer MongoDB.

**Getting Started with the mongo Shell (page 272)** Introduces the use and operation of the MongoDB shell.

**Access the mongo Shell Help Information (page 276)** Describes the available methods for accessing online help for the operation of the mongo interactive shell.

**mongo Shell Quick Reference (page 278)** A high level reference to the use and operation of the mongo shell.

---

**Server-side JavaScript**

**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)].

- `$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 266)].

---

**JavaScript in MongoDB**

Although these methods use JavaScript, most interactions with MongoDB do not use JavaScript but use an idiomatic driver in the language of the interacting application.

You can also disable server-side execution of JavaScript. For details, see [Disable Server-Side Execution of JavaScript (page 267)].

---

**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 278) 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

<table>
<thead>
<tr>
<th>Shell Helper</th>
<th>JavaScript Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>show dbs, show databases</td>
<td>db.adminCommand('listDatabases')</td>
</tr>
<tr>
<td>use &lt;db&gt;</td>
<td>db = db.getSiblingDB('&lt;db&gt;')</td>
</tr>
<tr>
<td>show collections</td>
<td>db.getCollectionNames()</td>
</tr>
<tr>
<td>show users</td>
<td>db.getUsers()</td>
</tr>
<tr>
<td>show roles</td>
<td>db.getRoles({showBuiltinRoles: true})</td>
</tr>
<tr>
<td>show log &lt;logname&gt;</td>
<td>db.adminCommand({'getLog': '&lt;logname&gt;'})</td>
</tr>
<tr>
<td>show logs</td>
<td>db.adminCommand({'getLog': '*'})</td>
</tr>
<tr>
<td>it</td>
<td>cursor = db.collection.find()</td>
</tr>
<tr>
<td></td>
<td>if (cursor.hasNext()) { cursor.next(); }</td>
</tr>
</tbody>
</table>

**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 759).

**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 235)

**Data Types in the *mongo* Shell**

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** 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:

```javascript
var myDateString = Date();
```

To print the value of the variable, type the variable name in the shell, as in the following:

```javascript
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:

```javascript
typeof myDateString
```

The operation returns `string`.

**Return Date** The mongo shell wrap 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.

```javascript
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:

```javascript
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:

```javascript
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 176) 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

You can write scripts for the `mongo` shell in JavaScript that manipulate data in MongoDB or perform administrative operations. For more information about the `mongo` shell see [MongoDB Scripting](page 265), and see the [Running .js files via a mongo shell Instance on the Server](page 266) 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 857) 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 73).
• 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.

<table>
<thead>
<tr>
<th>Shell Helpers</th>
<th>JavaScript Equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>show dbs, show databases</td>
<td>db.adminCommand('listDatabases')</td>
</tr>
<tr>
<td>use &lt;db&gt;</td>
<td>db = db.getSiblingDB('&lt;db&gt;')</td>
</tr>
<tr>
<td>show collections</td>
<td>db.getCollectionNames()</td>
</tr>
<tr>
<td>show users</td>
<td>db.getUsers()</td>
</tr>
<tr>
<td>show roles</td>
<td>db.getRoles({showBuiltInRoles: true})</td>
</tr>
<tr>
<td>show log &lt;logname&gt;</td>
<td>db.adminCommand({'getLog': '&lt;logname&gt;'})</td>
</tr>
<tr>
<td>show logs</td>
<td>db.adminCommand({'getLog': '*'})</td>
</tr>
<tr>
<td>it</td>
<td>cursor = db.collection.find()</td>
</tr>
<tr>
<td></td>
<td>if (cursor.hasNext()) {</td>
</tr>
<tr>
<td></td>
<td>cursor.next();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

• 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 mongod connection parameters inside of the javascript file using the Mongo() constructor. See Opening New Connections (page 270) for more information.

You can execute a .js file from within the mongo shell, using the load() function, as in the following:

```
load("myjstest.js")
```

This function loads and executes the myjstest.js file.

The load() method accepts relative and absolute paths. If the current working directory of the mongo shell is /data/db, and the myjstest.js resides in the /data/db/scripts directory, then the following calls within the mongo shell would be equivalent:

```
load("scripts/myjstest.js")
load("/data/db/scripts/myjstest.js")
```

**Note:** There is no search path for the load() function. If the desired script is not in the current working directory or the full specified path, mongo will not be able to access the file.

---

Getting Started with the mongo Shell

This document provides a basic introduction to using the mongo shell. See Install MongoDB (page 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 756) 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:

```javascript
db.<collection>.find()
```

- The `db` refers to the current database.
- The `<collection>` is the name of the collection to query. See Collection Help (page 277) 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:

```javascript
db["3test"].find()
```

```javascript
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:

```javascript
DBQuery.shellBatchSize = 10;
```

For more information and examples on cursor handling in the mongo shell, see Cursors (page 64). See also Cursor Help (page 277) 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
- mongo Shell Quick Reference (page 278)
- Read Operations (page 60)
- Write Operations (page 73)
- Indexing Tutorials (page 517)

### 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

---

108 http://docs.mongodb.org/getting-started/shell

---
Following:

```javascript
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:

```javascript
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:

```javascript
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:

```javascript
cmdCount = 1;
prompt = function() {
    return (cmdCount++) + " > ";
}
```

The prompt would then resemble the following:

```
1> db.collection.find()
2> show collections
3>
```
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 environment's EDITOR variable.

At your system prompt you can define the EDITOR variable and start mongo with the following two operations:

export EDITOR=vim
mongo

Then, consider the following example shell session:

MongoDB shell version: 2.2.0
> function f() {}
> edit f
> f
function f() {
    print("this really works");
}
> f()
this really works
> o = {}
{
} > edit o
> o
{ "soDoes" : "this" } >

Note: As mongo shell interprets code edited in an external editor, it may modify code in functions, depending on
the JavaScript compiler. For `mongo` may convert 1+1 to 2 or remove comments. The actual changes affect only the appearance of the code and will vary based on the version of JavaScript used but will not affect the semantics of the code.

**Exit the Shell**

To exit the shell, type `quit()` or use the `<Ctrl-c>` shortcut.

[See also: Getting Started Guide](#)

**Access the `mongo` Shell Help Information**

In addition to the documentation in the MongoDB Manual, the `mongo` shell provides some additional information in its “online” help system. This document provides an overview of accessing this help information.

[See also:](#)

- `mongo` Manual Page
- *MongoDB Scripting* (page 265), and
- *mongo Shell Quick Reference* (page 278).

**Command Line Help**

To see the list of options and help for starting the `mongo` shell, use the `--help` option from the command line:

```
mongo --help
```

**Shell Help**

To see the list of help in the `mongo` shell, type `help`:

```
help
```

**Database Help**

- To see the list of databases on the server, use the `show` `dbs` command:

  ```
  show dbs
  ```

  New in version 2.4: `show databases` is now an alias for `show dbs`

- To see the list of help for methods you can use on the `db` object, call the `db.help()` method:

  ```
  db.help()
  ```

- To see the implementation of a method in the shell, type the `db.<method name>` without the parenthesis `()`), as in the following example which will return the implementation of the method `db.updateUser()`:

  ```
  109
  ```

[109](https://docs.mongodb.org/getting-started/shell)
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 61) 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 64). See also `js-query-cursor-methods` for all available cursor methods.
Type Help

To get a list of the wrapper classes available in the mongo shell, such as BinData(), type help misc in the mongo shell:

```
help misc
```

mongo Shell Quick Reference

mongo Shell Command History

You can retrieve previous commands issued in the mongo shell with the up and down arrow keys. Command history is stored in ~/.dbshell file. See .dbshell for more information.

Command Line Options

The mongo shell can be started with numerous options. See mongo shell page for details on all available options. The following table displays some common options for mongo:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help</td>
<td>Show command line options</td>
</tr>
<tr>
<td>--nodb</td>
<td>Start mongo shell without connecting to a database.</td>
</tr>
<tr>
<td>--shell</td>
<td>Used in conjunction with a JavaScript file (i.e. &lt;file.js&gt;) to continue in the mongo shell after running the JavaScript file.</td>
</tr>
</tbody>
</table>

See JavaScript file (page 272) for an example.

Command Helpers

The mongo shell provides various help. The following table displays some common help methods and commands:

<table>
<thead>
<tr>
<th>Help Methods and Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Show help.</td>
</tr>
<tr>
<td>db.help()</td>
<td>Show help for database methods.</td>
</tr>
<tr>
<td>db.&lt;collection&gt;.help()</td>
<td>Show help on collection methods. The &lt;collection&gt; can be the name of an existing collection or a non-existing collection.</td>
</tr>
<tr>
<td>show dbs</td>
<td>Print a list of all databases on the server.</td>
</tr>
<tr>
<td>use &lt;db&gt;</td>
<td>Switch current database to &lt;db&gt;. The mongo shell variable db is set to the current database.</td>
</tr>
<tr>
<td>show collections</td>
<td>Print a list of all collections for current database</td>
</tr>
<tr>
<td>show users</td>
<td>Print a list of users for current database.</td>
</tr>
<tr>
<td>show roles</td>
<td>Print a list of all roles, both user-defined and built-in, for the current database.</td>
</tr>
<tr>
<td>show profile</td>
<td>Print the five most recent operations that took 1 millisecond or more. See documentation on the database profiler (page 227) for more information.</td>
</tr>
<tr>
<td>show databases</td>
<td>New in version 2.4: Print a list of all available databases.</td>
</tr>
<tr>
<td>load()</td>
<td>Execute a JavaScript file. See Getting Started with the mongo Shell (page 272) for more information.</td>
</tr>
</tbody>
</table>
**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:

<table>
<thead>
<tr>
<th>JavaScript Database Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.auth()</code></td>
<td>If running in secure mode, authenticate the user.</td>
</tr>
<tr>
<td><code>coll = db.&lt;collection&gt;</code></td>
<td>Set a specific collection in the current database to a variable <code>coll</code>, as in the following example: <code>coll = db.myCollection</code>; You can perform operations on the <code>myCollection</code> using the variable, as in the following example: <code>coll.find();</code></td>
</tr>
<tr>
<td><code>find()</code></td>
<td>Find all documents in the collection and returns a cursor. See the <code>db.collection.find()</code> and <code>Query Documents</code> (page 96) for more information and examples. See <code>Cursors</code> (page 64) for additional information on cursor handling in the <code>mongo</code> shell.</td>
</tr>
<tr>
<td><code>insert()</code></td>
<td>Insert a new document into the collection.</td>
</tr>
<tr>
<td><code>update()</code></td>
<td>Update an existing document in the collection. See <code>Write Operations</code> (page 73) for more information.</td>
</tr>
<tr>
<td><code>save()</code></td>
<td>Insert either a new document or update an existing document in the collection. See <code>Write Operations</code> (page 73) for more information.</td>
</tr>
<tr>
<td><code>remove()</code></td>
<td>Delete documents from the collection. See <code>Write Operations</code> (page 73) for more information.</td>
</tr>
<tr>
<td><code>drop()</code></td>
<td>Drops or removes completely the collection.</td>
</tr>
<tr>
<td><code>createIndex()</code></td>
<td>Create a new index on the collection if the index does not exist; otherwise, the operation has no effect. Return a reference to another database using this same connection without explicitly switching the current database. This allows for cross database queries. See <code>How can I access different databases temporarily?</code> (page 756) for more information.</td>
</tr>
<tr>
<td><code>db.getSiblingDB()</code></td>
<td></td>
</tr>
</tbody>
</table>

For more information on performing operations in the shell, see:

- *MongoDB CRUD Concepts* (page 60)
- *Read Operations* (page 60)
- *Write Operations* (page 73)
- *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:
<table>
<thead>
<tr>
<th>Keystroke</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up-arrow</td>
<td>previous-history</td>
</tr>
<tr>
<td>Down-arrow</td>
<td>next-history</td>
</tr>
<tr>
<td>Home</td>
<td>beginning-of-line</td>
</tr>
<tr>
<td>End</td>
<td>end-of-line</td>
</tr>
<tr>
<td>Tab</td>
<td>autocomplete</td>
</tr>
<tr>
<td>Left-arrow</td>
<td>backward-character</td>
</tr>
<tr>
<td>Right-arrow</td>
<td>forward-character</td>
</tr>
<tr>
<td>Ctrl-left-arrow</td>
<td>backward-word</td>
</tr>
<tr>
<td>Ctrl-right-arrow</td>
<td>forward-word</td>
</tr>
<tr>
<td>Meta-left-arrow</td>
<td>backward-word</td>
</tr>
<tr>
<td>Meta-right-arrow</td>
<td>forward-word</td>
</tr>
<tr>
<td>Ctrl-A</td>
<td>beginning-of-line</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>backward-char</td>
</tr>
<tr>
<td>Ctrl-C</td>
<td>exit-shell</td>
</tr>
<tr>
<td>Ctrl-D</td>
<td>delete-char (or exit shell)</td>
</tr>
<tr>
<td>Ctrl-E</td>
<td>end-of-line</td>
</tr>
<tr>
<td>Ctrl-F</td>
<td>forward-char</td>
</tr>
<tr>
<td>Ctrl-G</td>
<td>abort</td>
</tr>
<tr>
<td>Ctrl-J</td>
<td>accept-line</td>
</tr>
<tr>
<td>Ctrl-K</td>
<td>kill-line</td>
</tr>
<tr>
<td>Ctrl-L</td>
<td>clear-screen</td>
</tr>
<tr>
<td>Ctrl-M</td>
<td>accept-line</td>
</tr>
<tr>
<td>Ctrl-N</td>
<td>next-history</td>
</tr>
<tr>
<td>Ctrl-P</td>
<td>previous-history</td>
</tr>
<tr>
<td>Ctrl-R</td>
<td>reverse-search-history</td>
</tr>
<tr>
<td>Ctrl-S</td>
<td>forward-search-history</td>
</tr>
<tr>
<td>Ctrl-T</td>
<td>transpose-chars</td>
</tr>
<tr>
<td>Ctrl-U</td>
<td>unix-line-discard</td>
</tr>
<tr>
<td>Ctrl-W</td>
<td>unix-word-rubout</td>
</tr>
<tr>
<td>Ctrl-Y</td>
<td>yank</td>
</tr>
<tr>
<td>Ctrl-Z</td>
<td>Suspend (job control works in linux)</td>
</tr>
<tr>
<td>Ctrl-H (i.e. Backspace)</td>
<td>backward-delete-char</td>
</tr>
<tr>
<td>Ctrl-I (i.e. Tab)</td>
<td>complete</td>
</tr>
<tr>
<td>Meta-B</td>
<td>backward-word</td>
</tr>
<tr>
<td>Meta-C</td>
<td>capitalize-word</td>
</tr>
<tr>
<td>Meta-D</td>
<td>kill-word</td>
</tr>
<tr>
<td>Meta-F</td>
<td>forward-word</td>
</tr>
<tr>
<td>Meta-L</td>
<td>downcase-word</td>
</tr>
<tr>
<td>Meta-U</td>
<td>upcase-word</td>
</tr>
<tr>
<td>Meta-Y</td>
<td>yank-pop</td>
</tr>
<tr>
<td>Meta-[Backspace]</td>
<td>backward-kill-word</td>
</tr>
<tr>
<td>Meta-&lt;</td>
<td>beginning-of-history</td>
</tr>
<tr>
<td>Meta-&gt;</td>
<td>end-of-history</td>
</tr>
</tbody>
</table>

### 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

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find the documents matching the <code>&lt;query&gt;</code> criteria in the collection. If the <code>&lt;query&gt;</code> criteria is not specified or is empty (i.e., <code>{}</code>), the read operation selects all documents in the collection. The following example selects the documents in the <code>users</code> collection with the <code>name</code> field equal to &quot;Joe&quot;:</td>
<td><code>coll = db.users; coll.find( { name: &quot;Joe&quot; } );</code></td>
</tr>
<tr>
<td>Find documents matching the <code>&lt;query&gt;</code> criteria and return just specific fields in the <code>&lt;projection&gt;</code>..</td>
<td><code>coll = db.users; coll.find( { }, { name: true } );</code></td>
</tr>
<tr>
<td>Return results in the specified <code>&lt;sort order&gt;</code>..</td>
<td><code>coll = db.users; coll.find().sort( { name: 1 } );</code></td>
</tr>
<tr>
<td>Return the documents matching the <code>&lt;query&gt;</code> criteria in the specified <code>&lt;sort order&gt;</code>. Limit result to <code>&lt;n&gt;</code> rows. Highly recommended if you need only a certain number of rows for best performance. Skip <code>&lt;n&gt;</code> results.</td>
<td><code>coll = db.users; coll.find().sort( { name: 1 } ).limit( 10 )</code></td>
</tr>
<tr>
<td>Returns total number of documents in the collection. Returns the total number of documents that match the query. The <code>count()</code> ignores <code>limit()</code> and <code>skip()</code>. For example, if 100 records match but the limit is 10, <code>count()</code> will return 100. This will be faster than iterating yourself, but still take time. Find and return a single document. Returns null if not found.</td>
<td><code>count()</code></td>
</tr>
</tbody>
</table>

Internally, the `findOne()` method is the `find()` method with a `limit(1)`. For more information on specifying the `<query>` criteria, see [Query Documents](#). For more information on specifying the `<projection>`, see [Limit Fields to Return from a Query](#).

### Example

```javascript
coll = db.users;
coll.find( { name: "Joe" } );
```

### Example

```javascript
coll = db.users;
coll.find().sort( { name: 1 } );
```

### Example

```javascript
coll = db.users;
coll.findOne( { name: "Joe" } );
```
See *Query Documents* (page 96) and *Read Operations* (page 60) documentation for more information and examples. See [http://docs.mongodb.org/manual/reference/operator/query](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 78) 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:

<table>
<thead>
<tr>
<th>JavaScript Database Administration Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.cloneDatabase(&lt;host&gt;)</code></td>
<td>Clone the current database from the <code>&lt;host&gt;</code> specified. The <code>&lt;host&gt;</code> database instance must be in noauth mode.</td>
</tr>
<tr>
<td><code>db.copyDatabase(&lt;from&gt;, &lt;to&gt;, &lt;host&gt;)</code></td>
<td>Copy the <code>&lt;from&gt;</code> database from the <code>&lt;host&gt;</code> to the <code>&lt;to&gt;</code> database on the current server. The <code>&lt;host&gt;</code> database instance must be in noauth mode.</td>
</tr>
<tr>
<td><code>db.fromColl.renameCollection(&lt;toColl&gt;)</code></td>
<td>Rename collection from <code>fromColl</code> to <code>&lt;toColl&gt;</code>.</td>
</tr>
<tr>
<td><code>db.repairDatabase()</code></td>
<td>Repair and compact the current database. This operation can be very slow on large databases.</td>
</tr>
<tr>
<td><code>db.getCollectionNames()</code></td>
<td>Get the list of all collections in the current database.</td>
</tr>
<tr>
<td><code>db.dropDatabase()</code></td>
<td>Drops the current database.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>JavaScript Connection Create Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db = connect(&quot;&lt;host&gt;:&lt;port&gt;/&lt;dbname&gt;&quot;)</code></td>
<td>Open a new database connection.</td>
</tr>
<tr>
<td><code>conn = new Mongo()</code></td>
<td>Open a connection to a new server using <code>new Mongo()</code>.</td>
</tr>
<tr>
<td><code>db = conn.getDB(&quot;dbname&quot;)</code></td>
<td>Use <code>getDB()</code> method of the connection to select a database.</td>
</tr>
</tbody>
</table>

See also *Opening New Connections* (page 270) for more information on the opening new connections from the *mongo* shell.
Miscellaneous

The following table displays some miscellaneous methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object.bsonsize(&lt;document&gt;)</td>
<td>Prints the BSON size of a &lt;document&gt; in bytes</td>
</tr>
</tbody>
</table>

See the MongoDB JavaScript API Documentation\(^{110}\) for a full list of JavaScript methods.

Additional Resources

Consider the following reference material that addresses the mongo shell and its interface:

- \texttt{mongo}
- \texttt{js-administrative-methods}
- \texttt{database-commands}
- \textit{Aggregation Reference} (page 469)
- \texttt{Getting Started Guide}\(^{111}\)

Additionally, the MongoDB source code repository includes a jstests directory\(^{112}\) 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 tutorial in the manual, MongoDB provides \textit{Getting Started Guides} in various driver editions. If there is a process or pattern that you would like to see included here, please open a Jira Case\(^{113}\).

Installation

- \textit{Install MongoDB on Linux Systems} (page 22)
- \textit{Install MongoDB on Red Hat Enterprise or CentOS Linux} (page 7)
- \textit{Install MongoDB on Debian} (page 19)
- \textit{Install MongoDB on Ubuntu} (page 16)
- \textit{Install MongoDB on Amazon Linux} (page 13)
- \textit{Install MongoDB on SUSE} (page 10)
- \textit{Install MongoDB on OS X} (page 24)
- \textit{Install MongoDB on Windows} (page 27)

\(^{110}\)http://api.mongodb.org/js/index.html
\(^{111}\)https://docs.mongodb.org/getting-started/shell
\(^{112}\)https://github.com/mongodb/mongo/tree/master/jstests/
\(^{113}\)https://jira.mongodb.org/browse/DOCS
Administration

Replica Sets

- Deploy a Replica Set (page 603)
- Deploy Replica Set and Configure Authentication and Authorization (page 353)
- Convert a Standalone to a Replica Set (page 614)
- Add Members to a Replica Set (page 616)
- Remove Members from Replica Set (page 618)
- Replace a Replica Set Member (page 620)
- Adjust Priority for Replica Set Member (page 620)
- Resync a Member of a Replica Set (page 633)
- Deploy a Geographically Redundant Replica Set (page 608)
- Change the Size of the Oplog (page 628)
- Force a Member to Become Primary (page 631)
- Change Hostnames in a Replica Set (page 642)
- Add an Arbiter to Replica Set (page 614)
- Convert a Secondary to an Arbiter (page 626)
- Configure a Secondary’s Sync Target (page 645)
- Configure a Delayed Replica Set Member (page 624)
- Configure a Hidden Replica Set Member (page 623)
- Configure Non-Voting Replica Set Member (page 625)
- Prevent Secondary from Becoming Primary (page 621)
- Configure Replica Set Tag Sets (page 635)
- Manage Chained Replication (page 641)
- Reconfigure a Replica Set with Unavailable Members (page 638)
- Recover Data after an Unexpected Shutdown (page 263)
- Troubleshoot Replica Sets (page 646)

Sharding

- Deploy a Sharded Cluster (page 691)
- Convert a Replica Set to a Replicated Sharded Cluster (page 699)
- Add Shards to a Cluster (page 697)
- Remove Shards from an Existing Sharded Cluster (page 719)
- Deploy Three Config Servers for Production Deployments (page 698)
- Migrate Config Servers with the Same Hostname (page 707)
- Migrate Config Servers with Different Hostnames (page 708)
• Replace Disabled Config Server (page 709)
• Migrate a Sharded Cluster to Different Hardware (page 709)
• Backup Cluster Metadata (page 712)
• Backup a Small Sharded Cluster with mongodump (page 255)
• Backup a Sharded Cluster with Filesystem Snapshots (page 256)
• Backup a Sharded Cluster with Database Dumps (page 258)
• Restore a Single Shard (page 260)
• Restore a Sharded Cluster (page 260)
• Schedule Backup Window for Sharded Clusters (page 259)
• Manage Shard Tags (page 730)

Basic Operations

• Use Database Commands (page 223)
• Recover Data after an Unexpected Shutdown (page 263)
• Expire Data from Collections by Setting TTL (page 211)
• Analyze Performance of Database Operations (page 227)
• Rotate Log Files (page 231)
• Build Old Style Indexes (page 523)
• Manage mongod Processes (page 224)
• Back Up and Restore with MongoDB Tools (page 249)
• Backup and Restore with Filesystem Snapshots (page 243)

Security

• Configure Linux iptables Firewall for MongoDB (page 335)
• Configure Windows netsh Firewall for MongoDB (page 339)
• Enable Client Access Control (page 357)
• Create a User Administrator (page 384)
• Add a User to a Database (page 386)
• Create a Role (page 388)
• Modify a User’s Access (page 393)
• View Roles (page 394)
• Generate a Key File (page 379)
• Configure MongoDB with Kerberos Authentication on Linux (page 372)
• Create a Vulnerability Report (page 403)
Development Patterns

- Perform Two Phase Commits (page 115)
- Create an Auto-Incrementing Sequence Field (page 125)
- Enforce Unique Keys for Sharded Collections (page 732)
- Aggregation Examples (page 452)
- Model Data to Support Keyword Search (page 165)
- Limit Number of Elements in an Array after an Update (page 109)
- Perform Incremental Map-Reduce (page 463)
- Troubleshoot the Map Function (page 465)
- Troubleshoot the Reduce Function (page 466)
- Store a JavaScript Function on the Server (page 235)

Text Search Patterns

- Create a text Index (page 539)
- Specify a Language for Text Index (page 540)
- Specify Name for text Index (page 542)
- Control Search Results with Weights (page 543)
- Limit the Number of Entries Scanned (page 544)

Data Modeling Patterns

- Model One-to-One Relationships with Embedded Documents (page 152)
- Model One-to-Many Relationships with Embedded Documents (page 153)
- Model One-to-Many Relationships with Document References (page 154)
- Model Data for Atomic Operations (page 164)
- Model Tree Structures with Parent References (page 157)
- Model Tree Structures with Child References (page 158)
- Model Tree Structures with Materialized Paths (page 161)
- Model Tree Structures with Nested Sets (page 163)

See also:

The MongoDB Manual contains administrative documentation and tutorials through out several sections. See Replica Set Tutorials (page 602) and Sharded Cluster Tutorials (page 690) for additional tutorials and information.

5.3 Administration Reference

UNIX ulimit Settings (page 287) Describes user resources limits (i.e. ulimit) and introduces the considerations and optimal configurations for systems that run MongoDB deployments.
**System Collections** *(page 291)* Introduces the internal collections that MongoDB uses to track per-database metadata, including indexes, collections, and authentication credentials.

**Database Profiler Output** *(page 292)* Describes the data collected by MongoDB’s operation profiler, which introspects operations and reports data for analysis on performance and behavior.

**Server Status Output** *(page 296)* Provides an example and a high level overview of the output of the `serverStatus` command.

**Journaling Mechanics** *(page 308)* Describes the internal operation of MongoDB’s journaling facility and outlines how the journal allows MongoDB to provide durability and crash resiliency.

**Exit Codes and Statuses** *(page 310)* Lists the unique codes returned by `mongos` and `mongod` processes upon exit.

### 5.3.1 UNIX ulimit Settings

Most UNIX-like operating systems, including Linux and OS X, provide ways to limit and control the usage of system resources such as threads, files, and network connections on a per-process and per-user basis. These “ulimits” prevent single users from using too many system resources. Sometimes, these limits have low default values that can cause a number of issues in the course of normal MongoDB operation.

**Note:** Red Hat Enterprise Linux and CentOS 6 place a max process limitation of 1024 which overrides `ulimit` 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 211)*, 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 (kbytes) 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
- `-n` (open files): 64000
- `-m` (memory size): unlimited
- `-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.

Specify the **Recommended `ulimit` Settings** (page 289), as in the following example:

```plaintext
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 after changing `limit` stanzas, ensure that the changes take effect by restarting the application services, using the following form:

```
restart <service name>
```

---

114 If you limit virtual or resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

115 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.

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 directives117.

Specify the Recommended ulimit Settings (page 289), 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 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:

117http://www.freedesktop.org/software/systemd/man/systemd.exec.html#LimitCPU=
5.3.2 System Collections

Synopsis

MongoDB stores system information in collections that use the `<database>.system.*` namespace, which MongoDB reserves for internal use. Do not create collections that begin with `system`.

MongoDB also stores some additional instance-local metadata in the `local database` (page 652), 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 291) 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 291) 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 291) collection.

- `admin.system.version`  
  New in version 2.6.  
  Stores the schema version of the user credential documents.

System collections also include these collections stored directly in each database:

- `<database>.system.namespaces`  
  Deprecated since version 3.0: Access this data using `listCollections`.  
  The `<database>.system.namespaces` (page 291) collection contains information about all of the database’s collections.

- `<database>.system.indexes`  
  Deprecated since version 3.0: Access this data using `listIndexes`.  
  The `<database>.system.indexes` (page 291) collection lists all the indexes in the database.

- `<database>.system.profile`  
  The `<database>.system.profile` (page 291) collection stores database profiling information. For information on profiling, see `Database Profiling` (page 215).

- `<database>.system.js`  
  The `<database>.system.js` (page 291) collection holds special JavaScript code for use in `server side JavaScript` (page 266). See `Store a JavaScript Function on the Server` (page 235) for more information.
5.3.3 Database Profiler Output

The database profiler captures data information about read and write operations, cursor operations, and database commands. To configure the database profile and set the thresholds for capturing profile data, see the *Analyze Performance of Database Operations* (page 227) section.

The database profiler writes data in the `system.profile` (page 291) collection, which is a *capped collection*. To view the profiler’s output, use normal MongoDB queries on the `system.profile` (page 291) collection.

**Note:** Because the database profiler writes data to the `system.profile` (page 291) 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 291) collection have the following form. This example document reflects an insert operation:

```json
{
    "op" : "insert",
    "ns" : "test.orders",
    "query" : {
        "_id" : 1,
        "cust_id" : "A123",
        "amount" : 500,
        "status" : "A"
    },
    "ninserted" : 1,
    "keyUpdates" : 0,
    "writeConflicts" : 0,
    "numYield" : 0,
    "locks" : {
        "Global" : {
            "acquireCount" : {
                "w" : NumberLong(1)
            }
        },
        "MMAPV1Journal" : {
            "acquireCount" : {
                "w" : NumberLong(2)
            }
        },
        "Database" : {
            "acquireCount" : {
                "w" : NumberLong(1)
            }
        },
        "Collection" : {
            "acquireCount" : {
                "W" : NumberLong(1)
            }
        }
    },
    "millis" : 0,
    "execStats" : {
    },
    "ts" : ISODate("2012-12-10T19:31:28.977Z")
}
```
"client" : "127.0.0.1",
"allUsers" : [ ],
"user" : ""
}

**Output Reference**

For any single operation, the documents created by the database profiler will include a subset of the following fields. The precise selection of fields in these documents depends on the type of operation.

**Note:** For the output specific to the version of your MongoDB, refer to the appropriate version of the MongoDB Manual.

- **system.profile.op**
  The type of operation. The possible values are:
  - insert
  - query
  - update
  - remove
  - getmore
  - command

- **system.profile.ns**
  The namespace the operation targets. Namespaces in MongoDB take the form of the database, followed by a dot (.), followed by the name of the collection.

- **system.profile.query**
  The query document (page 96) used, or for an insert operation, the inserted document. If the document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (...). At the end of the string.

  Changed in version 3.0.4: For "getmore" (page 293) operations on cursors returned from a db.collection.find() or a db.collection.aggregate(), the query (page 293) field contains respectively the query predicate or the issued aggregate command document. For details on the aggregate command document, see the aggregate reference page.

- **system.profile.command**
  The command operation. If the command document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (...). At the end of the string.

- **system.profile.updateobj**
  The <update> document passed in during an update (page 102) operation. If the document exceeds 50 kilobytes, the value is a string summary of the object. If the string summary exceeds 50 kilobytes, the string summary is truncated, denoted with an ellipsis (...). At the end of the string.

- **system.profile.cursorid**
  The ID of the cursor accessed by a query and getmore operations.

- **system.profile.ntoreturn**
  The number of documents the operation specified to return. For example, the profile command would return one document (a results document) so the ntoreturn (page 293) value would be 1. The limit(5) command would return five documents so the ntoreturn (page 293) value would be 5.
If the `ntoreturn` (page 293) value is 0, the command did not specify a number of documents to return, as would be the case with a simple `find()` command with no limit specified.

**system.profile.ntoskip**
- The number of documents the `skip()` method specified to skip.

**system.profile.nscanned**
- The number of documents that MongoDB scans in the `index` (page 481) in order to carry out the operation.
- In general, if `nscanned` (page 294) is much higher than `nreturned` (page 295), the database is scanning many objects to find the target objects. Consider creating an index to improve this.

**system.profile.nscannedObjects**
- The number of documents that MongoDB scans from the collection in order to carry out the operation.

**system.profile.moved**
- Changed in version 3.0.0: Only appears when using the MMAPv1 storage engine.
- This field appears with a value of true when an update operation moved one or more documents to a new location on disk. If the operation did not result in a move, this field does not appear. Operations that result in a move take more time than in-place updates and typically occur as a result of document growth.

**system.profile.nmoved**
- Changed in version 3.0.0: Only appears when using the MMAPv1 storage engine.
- The number of documents the operation moved on disk. This field appears only if the operation resulted in a move. The field’s implicit value is zero, and the field is present only when non-zero.

**system.profile.scanAndOrder**
- `scanAndOrder` (page 294) is a boolean that is true when a query cannot use the ordering in the index to return the requested sorted results; i.e. MongoDB must sort the documents after it receives the documents from a cursor. The field only appears when the value is true.

**system.profile.ndeleted**
- The number of documents deleted by the operation.

**system.profile.ninserted**
- The number of documents inserted by the operation.

**system.profile.nMatched**
- New in version 2.6.
- The number of documents that match the `system.profile.query` (page 293) condition for the update operation.

**system.profile.nModified**
- New in version 2.6.
- The number of documents modified by the update operation.

**system.profile.upsert**
- A boolean that indicates the update operation’s upsert option value. Only appears if upsert is true.

**system.profile.keyUpdates**
- The number of `index` (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.writeConflicts**
- New in version 3.0.0.
- The number of conflicts encountered during the write operation; e.g. an update operation attempts to modify the same document as another update operation. See also write conflict.
system.profile.numYield

The number of times the operation yielded to allow other operations to complete. Typically, operations yield when they need access to data that MongoDB has not yet fully read into memory. This allows other operations that have data in memory to complete while MongoDB reads in data for the yielding operation. For more information, see the FAQ on when operations yield (page 759).

system.profile.locks

New in version 3.0.0: locks (page 295) replaces the lockStats field.

The system.profile.locks (page 295) provides information for various lock types and lock modes (page 758) held during the operation.

The possible lock types are:

• Global represents global lock.
• MMAPV1Journal represents MMAPv1 storage engine specific lock to synchronize journal writes; for non-MMAPv1 storage engines, the mode for MMAPV1Journal is empty.
• Database represents database lock.
• Collection represents collection lock.
• Metadata represents metadata lock.
• oplog represents lock on the oplog.

The possible locking modes for the lock types are as follows:

• R represents Shared (S) lock.
• W represents Exclusive (X) lock.
• r represents Intent Shared (IS) lock.
• w represents Intent Exclusive (IX) lock.

The returned lock information for the various lock types include:

system.profile.locks.acquireCount
Number of times the operation acquired the lock in the specified mode.

system.profile.locks.acquireWaitCount
Number of times the operation had to wait for the acquireCount (page 295) lock acquisitions because the locks were held in a conflicting mode. acquireWaitCount (page 295) is less than or equal to acquireCount (page 295).

system.profile.locks.timeAcquiringMicros
Cumulative time in microseconds that the operation had to wait to acquire the locks.

timeAcquiringMicros (page 295) divided by acquireWaitCount (page 295) gives an approximate average wait time for the particular lock mode.

system.profile.locks.deadlockCount
Number of times the operation encountered deadlocks while waiting for lock acquisitions.

For more information on lock modes, see What type of locking does MongoDB use? (page 758).

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 295) can affect performance. To limit the size of the result document for a query operation, you can use any of the following:

• Projections (page 107)
• The limit() method
• The batchSize() method

Note: When MongoDB writes query profile information to the log, the responseLength (page 295) value is in a field named reslen.

system.profile.millis
The time in milliseconds from the perspective of the mongod from the beginning of the operation to the end of the operation.

system.profile.execStats
Changed in version 3.0.
A document that contains the execution statistics of the query operation. For other operations, the value is an empty document.
The system.profile.execStats (page 296) presents the statistics as a tree; each node provides the statistics for the operation executed during that stage of the query operation.

Note: The following fields list for execStats (page 296) is not meant to be exhaustive as the returned fields vary per stage.

system.profile.execStats.stage
New in version 3.0: stage (page 296) replaces the type field.
The descriptive name for the operation performed as part of the query execution; e.g.
• COLLSCAN for a collection scan
• IXSCAN for scanning index keys
• FETCH for retrieving documents

system.profile.execStats.inputStages
New in version 3.0: inputStages (page 296) replaces the children field.
An array that contains statistics for the operations that are the input stages of the current stage.

system.profile.ts
The timestamp of the operation.

system.profile.client
The IP address or hostname of the client connection where the operation originates.
For some operations, such as db.eval(), the client is 0.0.0.0:0 instead of an actual client.

system.profile.allUsers
An array of authenticated user information (user name and database) for the session. See also Client Authentication (page 323).

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 Server Status Output

This document provides a quick overview and example of the serverStatus command. The helper db.serverStatus() in the mongo shell provides access to this output. For full documentation of the content of this output, see http://docs.mongodb.org/manual/reference/command/serverStatus.
Note: The output fields vary depending on the version of MongoDB, underlying operating system platform, the storage engine, and the kind of node, including mongos, mongod or replica set member. For the serverStatus output specific to the version of your MongoDB, refer to the appropriate version of the MongoDB Manual.

Changed in version 3.0: The server status output no longer includes the workingSet, indexCounters, and recordStats sections. The server-status-instance-information section displays information regarding the specific mongod and mongos and its state.

"host" : "<hostname>",
"version" : "<version>",
"process" : "<mongod|mongos>",
"pid" : <num>,
"uptime" : <num>,
"uptimeMillis" : <num>,
"uptimeEstimate" : <num>,
"localTime" : ISODate(""),

The server-status-asserts document reports the number of assertions or errors produced by the server:

"asserts" : {
  "regular" : <num>,
  "warning" : <num>,
  "msg" : <num>,
  "user" : <num>,
  "rollovers" : <num>
},

The server-status-backgroundflushing document reports on the process MongoDB uses to write data to disk. The server-status-backgroundflushing information only returns for instances that use the MMAPv1 storage engine:

"backgroundFlushing" : {
  "flushes" : <num>,
  "total_ms" : <num>,
  "average_ms" : <num>,
  "last_ms" : <num>,
  "last_finished" : ISODate(""")
},

The server-status-connections field reports on MongoDB’s current number of open incoming connections:

New in version 2.4: The totalCreated field.

"connections" : {
  "current" : <num>,
  "available" : <num>,
  "totalCreated" : NumberLong(<num>)
},

The server-status-cursors document reports on current cursor use and state:

"cursors" : {
  "note" : "deprecated, use server status metrics",
  "clientCursors_size" : <num>,
  "totalOpen" : <num>,
  "pinned" : <num>,
  "totalNoTimeout" : <num>,
  "timedOut" : <num>
},
The `server-status-journaling` document reports on data that reflect this `mongod` instance’s journaling-related operations and performance during a journal group commit interval (page 235). The `server-status-journaling` information only returns for instances that use the MMAPv1 storage engine and have journaling enabled:

```
"dur" : {
    "commits" : <num>,
    "journaledMB" : <num>,
    "writeToDataFilesMB" : <num>,
    "compression" : <num>,
    "commitsInWriteLock" : <num>,
    "earlyCommits" : <num>,
    "timeMs" : {
        "dt" : <num>,
        "prepLogBuffer" : <num>,
        "writeToJournal" : <num>,
        "writeToDataFiles" : <num>,
        "remapPrivateView" : <num>,
        "commits" : <num>,
        "commitsInWriteLock" : <num>
    }
},
```

The fields in the `server-status-extra-info` document provide platform specific information. The following example block is from a Linux-based system:

```
"extra_info" : {
    "note" : "fields vary by platform",
    "heap_usage_bytes" : <num>,
    "page_faults" : <num>
},
```

The `server-status-globallock` field reports on MongoDB’s global system lock. In most cases the `locks` document provides more fine grained data that reflects lock use:

```
"globalLock" : {
    "totalTime" : <num>,
    "currentQueue" : {
        "total" : <num>,
        "readers" : <num>,
        "writers" : <num>
    },
    "activeClients" : {
        "total" : <num>,
        "readers" : <num>,
        "writers" : <num>
    }
},
```

The `server-status-locks` section reports statistics for each lock type and mode:

```
"locks" : {
    "Global" : {
        "acquireCount" : {
            "r" : NumberLong(<num>),
            "w" : NumberLong(<num>),
            "R" : NumberLong(<num>),
            "W" : NumberLong(<num>)
        },
        "acquireWaitCount" : {
            "r" : NumberLong(<num>),
```
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"deadlockCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
}
},
"Collection" : {
"acquireCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"acquireWaitCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"timeAcquiringMicros" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"deadlockCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
}
},
"Metadata" : {
"acquireCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"acquireWaitCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"timeAcquiringMicros" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
},
"deadlockCount" : {
"r" : NumberLong(<num>),
"w" : NumberLong(<num>),
"R" : NumberLong(<num>),
"W" : NumberLong(<num>)
}
The `server-status-network` document reports on network use and state:

```json
"network" : {
    "bytesIn" : <num>,
    "bytesOut" : <num>,
    "numRequests" : <num>
}
```

The `server-status-opcounters` document reports the number of operations this MongoDB instance has processed:

```json
"opcounters" : {
    "insert" : <num>,
    "query" : <num>,
    "update" : <num>,
    "delete" : <num>,
    "getmore" : <num>,
    "command" : <num>
}
```

The `server-status-opcounters-repl` document reports the number of replicated operations:

```json
"opcountersRepl" : {
    "insert" : <num>,
    "query" : <num>,
    "update" : <num>,
    "delete" : <num>
}
```
The server-status-storage-engine document reports details about the current storage engine:

```
"storageEngine" : { 
   "name" : <string> 
}
```

The server-status-writebacksqueued document reports the number of writebacks:

```
"writeBacksQueued" : <num>,
```

The server-status-memory field reports on MongoDB’s current memory use:

```
"mem" : { 
   "bits" : <num>,
   "resident" : <num>,
   "virtual" : <num>,
   "supported" : <boolean>,
   "mapped" : <num>,
   "mappedWithJournal" : <num>,
   "note" : "not all mem info support on this platform"
}
```

The server-status-repl document reports on the state of replication and the replica set. This document only appears for replica sets.

```
"repl" : { 
   "setName" : <string>,
   "setVersion" : <num>,
   "ismaster" : <boolean>,
   "secondary" : <boolean>,
   "hosts" : [ 
      <hostname>,
      <hostname>,
      <hostname>
   ],
   "primary" : <hostname>,
   "me" : <hostname>,
   "electionId" : ObjectId(""),
   "rbid" : <num>,
   "slaves" : [ 
      { 
         "rid" : <ObjectId>,
         "optime" : <timestamp>,
         "host" : <hostname>,
         "memberID" : <num>
      }
   ],
}
```

The server-status-range-deleter document reports the number of operations this MongoDB instance has processed. The rangeDeleter document is only present in the output of serverStatus when explicitly enabled.

```
"rangeDeleter" : { 
   "lastDeleteStats" : [ 
      { 
         "deletedDocs" : NumberLong(<num>),
      }
   ],
}
```
The server-status-security document reports details about the security features and use:

```
"security" : {
   "SSLServerSubjectName": <string>,
   "SSLServerHasCertificateAuthority": <boolean>,
   "SSLServerCertificateExpirationDate": <date>
},
```

The server-status-metrics document contains a number of operational metrics that are useful for monitoring the state and workload of a mongod instance.

New in version 2.4.

Changed in version 2.6: Added the cursor document.

```
"metrics" : {
   "command": {
      "<command>": {
         "failed": <num>,
         "total": <num>
      }
   },
   "cursor" : {
      "timedOut" : NumberLong(<num>),
      "open" : {
         "noTimeout" : NumberLong(<num>),
         "pinned" : NumberLong(<num>),
         "multiTarget" : NumberLong(<num>),
         "singleTarget" : NumberLong(<num>),
         "total" : NumberLong(<num>),
      }
   },
   "document" : {
      "deleted" : NumberLong(<num>),
      "inserted" : NumberLong(<num>),
      "returned" : NumberLong(<num>),
      "updated" : NumberLong(<num>)
   },
   "getLastError" : {
      "wtime" : {
         "num": <num>,
         "totalMillis": <num>
      },
      "wtimeouts" : NumberLong(<num>)
   },
   "operation" : {
      "fastmod" : NumberLong(<num>),
      "idhack" : NumberLong(<num>),
      "scanAndOrder" : NumberLong(<num>)
   },
}```
"queryExecutor": {
    "scanned": NumberLong(<num>)
},
"record": {
    "moves": NumberLong(<num>)
},
"repl": {
    "apply": {
        "batches": {
            "num": <num>,
            "totalMillis": <num>
        },
        "ops": NumberLong(<num>)
    },
    "buffer": {
        "count": NumberLong(<num>),
        "maxSizeBytes": <num>,
        "sizeBytes": NumberLong(<num>)
    },
    "network": {
        "bytes": NumberLong(<num>),
        "getmores": {
            "num": <num>,
            "totalMillis": <num>
        },
        "ops": NumberLong(<num>),
        "readersCreated": NumberLong(<num>)
    },
    "oplog": {
        "insert": {
            "num": <num>,
            "totalMillis": <num>
        },
        "insertBytes": NumberLong(<num>)
    },
    "preload": {
        "docs": {
            "num": <num>,
            "totalMillis": <num>
        },
        "indexes": {
            "num": <num>,
            "totalMillis": <num>
        }
    }
},
"storage": {
    "freelist": {
        "search": {
            "bucketExhausted": <num>,
            "requests": <num>,
            "scanned": <num>
        }
    }
},
"ttl": {
    "deletedDocuments": NumberLong(<num>),
    "passes": NumberLong(<num>)
}
The `server-status-wiredTiger` statistics section reports details about the WiredTiger statistics:

New in version 3.0: `server-status-wiredTiger` statistics section. This section appears only for the WiredTiger storage engine.

```json
"wiredTiger": {
    "uri": "statistics:",
    "LSM": {
        "sleep for LSM checkpoint throttle": <num>,
        "sleep for LSM merge throttle": <num>,
        "rows merged in an LSM tree": <num>,
        "application work units currently queued": <num>,
        "merge work units currently queued": <num>,
        "tree queue hit maximum": <num>,
        "switch work units currently queued": <num>,
        "tree maintenance operations scheduled": <num>,
        "tree maintenance operations discarded": <num>,
        "tree maintenance operations executed": <num>
    },
    "async": {
        "number of allocation state races": <num>,
        "number of operation slots viewed for allocation": <num>,
        "current work queue length": <num>,
        "number of flush calls": <num>,
        "number of times operation allocation failed": <num>,
        "maximum work queue length": <num>,
        "number of times worker found no work": <num>,
        "total allocations": <num>,
        "total compact calls": <num>,
        "total insert calls": <num>,
        "total remove calls": <num>,
        "total search calls": <num>,
        "total update calls": <num>
    },
    "block-manager": {
        "mapped bytes read": <num>,
        "bytes read": <num>,
        "bytes written": <num>,
        "mapped blocks read": <num>,
        "blocks pre-loaded": <num>,
        "blocks read": <num>,
        "blocks written": <num>
    },
    "cache": {
        "tracked dirty bytes in the cache": <num>,
        "bytes currently in the cache": <num>,
        "maximum bytes configured": <num>,
        "bytes read into cache": <num>,
        "bytes written from cache": <num>,
        "pages evicted by application threads": <num>,
        "checkpoint blocked page eviction": <num>,
        "unmodified pages evicted": <num>,
        "page split during eviction deepened the tree": <num>,
        "modified pages evicted": <num>,
        "pages selected for eviction unable to be evicted": <num>,
        "pages evicted because they exceeded the in-memory maximum": <num>
    }
}
```
"pages evicted because they had chains of deleted items" : <num>,
"failed eviction of pages that exceeded the in-memory maximum" : <num>,
"hazard pointer blocked page eviction" : <num>,
"internal pages evicted" : <num>,
"maximum page size at eviction" : <num>,
"eviction server candidate queue empty when topping up" : <num>,
"eviction server candidate queue not empty when topping up" : <num>,
"eviction server evicting pages" : <num>,
"eviction server populating queue, but not evicting pages" : <num>,
"eviction server unable to reach eviction goal" : <num>,
"pages split during eviction" : <num>,
"pages walked for eviction" : <num>,
"eviction worker thread evicting pages" : <num>,
"in-memory page splits" : <num>,
"percentage overhead" : <num>,
"tracked dirty pages in the cache" : <num>,
"pages currently held in the cache" : <num>,
"pages read into cache" : <num>,
"pages written from cache" : <num>
},
"connection" : {
"pthread mutex condition wait calls" : <num>,
"files currently open" : <num>,
"memory allocations" : <num>,
"memory frees" : <num>,
"memory re-allocations" : <num>,
"total read I/Os" : <num>,
"pthread mutex shared lock read-lock calls" : <num>,
"pthread mutex shared lock write-lock calls" : <num>,
"total write I/Os" : <num>
},
"cursor" : {
"cursor create calls" : <num>,
"cursor insert calls" : <num>,
"cursor next calls" : <num>,
"cursor prev calls" : <num>,
"cursor remove calls" : <num>,
"cursor reset calls" : <num>,
"cursor search calls" : <num>,
"cursor search near calls" : <num>,
"cursor update calls" : <num>
},
"data-handle" : {
"connection dhandles swept" : <num>,
"connection candidate referenced" : <num>,
"connection sweeps" : <num>,
"connection time-of-death sets" : <num>,
"session dhandles swept" : <num>,
"session sweep attempts" : <num>
},
"log" : {
"log buffer size increases" : <num>,
"total log buffer size" : <num>,
"log bytes of payload data" : <num>,
"log bytes written" : <num>,
"yields waiting for previous log file close" : <num>,
"total size of compressed records" : <num>,
"total in-memory size of compressed records" : <num>,
"log records too small to compress" : <num>,
"log records not compressed" : <num>,
"log records compressed" : <num>,
"maximum log file size" : <num>,
"pre-allocated log files prepared" : <num>,
"number of pre-allocated log files to create" : <num>,
"pre-allocated log files used" : <num>,
"log read operations" : <num>,
"log release advances write LSN" : <num>,
"records processed by log scan" : <num>,
"log scan records requiring two reads" : <num>,
"log scan operations" : <num>,
"consolidated slot closures" : <num>,
"logging bytes consolidated" : <num>,
"consolidated slot joins" : <num>,
"consolidated slot join races" : <num>,
"slots selected for switching that were unavailable" : <num>,
"record size exceeded maximum" : <num>,
"failed to find a slot large enough for record" : <num>,
"consolidated slot join transitions" : <num>,
"log sync operations" : <num>,
"log sync_dir operations" : <num>,
"log server thread advances write LSN" : <num>,
"log write operations" : <num>
},
"reconciliation" : {
  "page reconciliation calls" : <num>,
  "page reconciliation calls for eviction" : <num>,
  "split bytes currently awaiting free" : <num>,
  "split objects currently awaiting free" : <num>
},
"session" : {
  "open cursor count" : <num>,
  "open session count" : <num>
},
"thread-yield" : {
  "page acquire busy blocked" : <num>,
  "page acquire eviction blocked" : <num>,
  "page acquire locked blocked" : <num>,
  "page acquire read blocked" : <num>,
  "page acquire time sleeping (usecs)" : <num>
},
"transaction" : {
  "transaction begins" : <num>,
  "transaction checkpoints" : <num>,
  "transaction checkpoint currently running" : <num>,
  "transaction checkpoint max time (msecs)" : <num>,
  "transaction checkpoint min time (msecs)" : <num>,
  "transaction checkpoint most recent time (msecs)" : <num>,
  "transaction checkpoint total time (msecs)" : <num>,
  "transactions committed" : <num>,
  "transaction failures due to cache overflow" : <num>,
  "transaction range of IDs currently pinned" : <num>,
  "transactions rolled back" : <num>
},
"concurrentTransactions" : {
  "write" : {
    "out" : <num>,
  }
}
The final `ok` field holds the return status for the `serverStatus` command:

```
"ok" : 1
```

### 5.3.5 Journaling Mechanics

When running with journaling, MongoDB stores and applies write operations (page 73) 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 233) 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 234).
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 60). The private view is the first place MongoDB applies new write operations (page 73). 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 73) 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.
5.3.6 Exit Codes and Statuses

MongoDB will return one of the following codes and statuses when exiting. Use this guide to interpret logs and when troubleshooting issues with mongod and mongos instances.

0
  Returned by MongoDB applications upon successful exit.

2
  The specified options are in error or are incompatible with other options.

3
  Returned by mongod if there is a mismatch between hostnames specified on the command line and in the local.sources (page 654) 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.
5.4 Production Checklist

The following checklists provide recommendations that will help you avoid issues in your production MongoDB deployment.

5.4.1 Operations Checklist

The following checklist, along with the Development (page 314) list, provides recommendations to help you avoid issues in your production MongoDB deployment.

Filesystem

- Align your disk partitions with your RAID configuration.
- Avoid using NFS drives for your `dbPath`. Using NFS drives can result in degraded and unstable performance. See: Remote Filesystems (page 201) for more information.
  - VMWare users should use VMWare virtual drives over NFS.
- Linux/Unix: format your drives into XFS or EXT4. If possible, use XFS as it generally performs better with MongoDB.
  - With the WiredTiger storage engine, use of XFS is strongly recommended to avoid performance issues found when using EXT4 with WiredTiger.
  - If using RAID, you may need to configure XFS with your RAID geometry.
- Windows: use the NTFS file system. Do not use any FAT file system (i.e. FAT 16/32/exFAT).

Replication

- Verify that all non-hidden replica set members are identically provisioned in terms of their RAM, CPU, disk, network setup, etc.
- Configure the oplog size (page 628) to suit your use case:
  - The replication oplog window should cover normal maintenance and downtime windows to avoid the need for a full resync.
  - The replication oplog window should cover the time needed to restore a replica set member, either by an initial sync or by restoring from the last backup.
- Ensure that your replica set includes at least three data-bearing nodes with `w:majority` `write concern` (page 130). Three data-bearing nodes are required for replica set-wide data durability.
- Use hostnames when configuring replica set members, rather than IP addresses.
- Ensure full bidirectional network connectivity between all `mongod` instances.
- Ensure that each host can resolve itself.
- Ensure that your replica set contains an odd number of voting members.
- Ensure that `mongod` instances have 0 or 1 votes.
- For high availability, deploy your replica set into a minimum of three data centers.
Sharding

- Place your config servers (page 670) on dedicated hardware for optimal performance in large clusters. Ensure that the hardware has enough RAM to hold the data files entirely in memory and that it has dedicated storage.
- Use NTP to synchronize the clocks on all components of your sharded cluster.
- Ensure full bidirectional network connectivity between mongod, mongos and config servers.
- Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.

Journaling: MMAPv1 Storage Engine

- Ensure that all instances use journaling (page 308).
- Place the journal on its own low-latency disk for write-intensive workloads. Note that this will affect snapshot-style backups as the files constituting the state of the database will reside on separate volumes.

Hardware

- Use RAID10 and SSD drives for optimal performance.
- SAN and Virtualization:
  - Ensure that each mongod has provisioned IOPS for its dbPath, or has its own physical drive or LUN.
  - Avoid dynamic memory features, such as memory ballooning, when running in virtual environments.
  - Avoid placing all replica set members on the same SAN, as the SAN can be a single point of failure.

Deployments to Cloud Hardware

- Windows Azure: Adjust the TCP keepalive (tcp_keepalive_time) to 100-120. The default TTL for TCP connections on Windows Azure load balancers is too slow for MongoDB’s connection pooling behavior.
- Use MongoDB version 2.6.4 or later on systems with high-latency storage, such as Windows Azure, as these versions include performance improvements for those systems. See: Azure Deployment Recommendations for more information.

Operating System Configuration

Linux

- Turn off transparent hugepages and defrag. See Transparent Huge Pages Settings (page 221) for more information.
- Adjust the readahead settings (page 203) on the devices storing your database files to suit your use case. If your working set is bigger than the available RAM, and the document access pattern is random, consider lowering the readahead to 32 or 16. Evaluate different settings to find an optimal value that maximizes the resident memory and lowers the number of page faults.
- Use the noop or deadline disk schedulers for SSD drives.
- Use the noop disk scheduler for virtualized drives in guest VMs.

118 https://docs.mongodb.org/ecosystem/platforms/windows-azure
• Disable NUMA or set `vm.zone_reclaim_mode` to 0 and run `mongod` instances with node interleaving. See: *MongoDB and NUMA Hardware* (page 200) for more information.

• Adjust the `ulimit` values on your hardware to suit your use case. If multiple `mongod` or `mongos` instances are running under the same user, scale the `ulimit` values accordingly. See: *UNIX ulimit Settings* (page 287) for more information.

• Use `noatime` for the `dbPath` mount point.

• Configure sufficient file handles (`fs.file-max`), kernel pid limit (`kernel.pid_max`), and maximum threads per process (`kernel.threads-max`) for your deployment. For large systems, values of 98000, 32768, and 64000 are a good starting point.

• Ensure that your system has swap space configured. Refer to your operating system’s documentation for details on appropriate sizing.

• Ensure that the system default TCP keepalive is set correctly. A value of 300 often provides better performance for replica sets and sharded clusters. See: *Does TCP keepalive time affect MongoDB Deployments?* (page 779) in the Frequently Asked Questions for more information.

**Windows**

• Consider disabling NTFS “last access time” updates. This is analogous to disabling `atime` on Unix-like systems.

**Backups**

• Schedule periodic tests of your back up and restore process to have time estimates on hand, and to verify its functionality.

**Monitoring**

• Use MongoDB Cloud Manager\(^{119}\) or Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced\(^{120}\) or another monitoring system to monitor key database metrics and set up alerts for them. Include alerts for the following metrics:
  – lock percent (for the *MMAPv1 storage engine* (page 91))
  – replication lag
  – replication oplog window
  – assertions
  – queues
  – page faults

• Monitor hardware statistics for your servers. In particular, pay attention to the disk use, CPU, and available disk space.

  In the absence of disk space monitoring, or as a precaution:
  – Create a dummy 4GB file on the `storage.dbPath` drive to ensure available space if the disk becomes full.

\(^{119}\)https://cloud.mongodb.com/?jmp=docs

\(^{120}\)https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
A combination of `cron+df` can alert when disk space hits a high-water mark, if no other monitoring tool is available.

**Load Balancing**

- Configure load balancers to enable “sticky sessions” or “client affinity”, with a sufficient timeout for existing connections.
- Avoid placing load balancers between MongoDB cluster or replica set components.

### 5.4.2 Development

The following checklist, along with the *Operations Checklist* (page 311), provides recommendations to help you avoid issues in your production MongoDB deployment.

**Data Durability**

- Ensure that your replica set includes at least three data-bearing nodes with `w:majority write concern` (page 130). Three data-bearing nodes are required for replica-set wide data durability.
- Ensure that all instances use `journaling` (page 308).

**Schema Design**

- Ensure that your schema design does not rely on indexed arrays that grow in length without bound. Typically, best performance can be achieved when such indexed arrays have fewer than 1000 elements.

**Replication**

- Do not use secondary reads to scale overall read throughput. See: Can I use more replica nodes to scale for an overview of read scaling. For information about secondary reads, see: *Read Preference* (page 588).

**Sharding**

- Ensure that your shard key distributes the load evenly on your shards. See: *Considerations for Selecting Shard Keys* (page 694) for more information.
- Use targeted queries (page 678) for workloads that need to scale with the number of shards.
- Always read from primary nodes for non-targeted queries that may be sensitive to stale or orphaned data.
- *Pre-split and manually balance chunks* (page 722) when inserting large data sets into a new non-hashed sharded collection. Pre-splitting and manually balancing enables the insert load to be distributed among the shards, increasing performance for the initial load.

---

Drivers

- Make use of connection pooling. Most MongoDB drivers support connection pooling. Adjust the connection pool size to suit your use case, beginning at 110-115% of the typical number of concurrent database requests.
- Ensure that your applications handle transient write and read errors during replica set elections.
- Ensure that your applications handle failed requests and retry them if applicable. Drivers do not automatically retry failed requests.
- Use exponential backoff logic for database request retries.
- Use `cursor.maxTimeMS()` for reads and `wtimeout` (page 131) for writes if you need to cap execution time for database operations.

5.4.3 Additional Resources

- MongoDB Production Readiness Consulting Package
- MongoDB Ops Optimization Consulting Package

---

123https://www.mongodb.com/products/consulting?jmp=docs#s_production_readiness
124https://www.mongodb.com/products/consulting?jmp=docs#ops_optimization
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 Concepts (page 320) The core documentation of security.

Authentication (page 320) Mechanisms for verifying user and instance access to MongoDB.

Authorization (page 324) Control access to MongoDB instances using authorization.

Network Exposure and Security (page 327) 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 320) for additional documentation of MongoDB’s security features and operation.

Security Tutorials (page 334) Tutorials for enabling and configuring security features for MongoDB.

Network Security Tutorials (page 335) 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 356) These tutorials describe procedures relevant for the configuration, operation, and maintenance of MongoDB’s access control system.

User and Role Management Tutorials (page 384) 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 and setup of the authorization system.

Continue reading from Security Tutorials (page 334) for additional tutorials that address the use and management of secure MongoDB deployments.

Create a Vulnerability Report (page 403) Report a vulnerability in MongoDB.

Security Reference (page 404) 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

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 321) 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 323) and Kerberos authentication (page 322).

See Authentication (page 320) for more information.

6.1.2 Role Based Access Control

Access control, i.e., authorization (page 324), 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 324), 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 324) 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.

See Auditing (page 330) 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 342) for more information.

\(^1\)http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
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.

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\(^8\)

---

\(^7\) 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
6.2 Security Concepts

These documents introduce and address concepts and strategies related to security practices in MongoDB deployments.

**Authentication** (page 320) Mechanisms for verifying user and instance access to MongoDB.

**Authorization** (page 324) Control access to MongoDB instances using authorization.

**Collection-Level Access Control** (page 326) Scope privileges to specific collections.

**Network Exposure and Security** (page 327) 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 329) Discusses potential risks related to MongoDB’s JavaScript, HTTP and REST interfaces, including strategies to control those risks.

**Auditing** (page 330) Audit server and client activity for mongod and mongos instances.

**Kerberos Authentication** (page 331) Kerberos authentication and MongoDB.

6.2.1 Authentication

Authentication is the process of verifying the identity of a client. When access control, i.e. **authorization** (page 324), is enabled, MongoDB requires all clients to authenticate themselves first in order to determine the access for the client.

Although authentication and **authorization** (page 324) 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 321) that clients can use to verify their identity. These mechanisms allow MongoDB to integrate into your existing authentication system. See **Authentication Mechanisms** (page 321) for details.

In addition to verifying the identity of a client, MongoDB can require members of replica sets and sharded clusters to **authenticate their membership** (page 323) to their respective replica set or sharded cluster. See **Authentication Between MongoDB Instances** (page 323) 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 386).

MongoDB stores all user information, including name (page 416), password (page 416), and the user’s database (page 416), in the `system.users` (page 416) collection in the `admin` database.

---

9http://www.mongodb.com/presentations/webinar-securing-your-mongodb-deployment?jmp=docs
Authentication Mechanisms

Changed in version 3.0.

MongoDB supports multiple authentication mechanisms. MongoDB’s default authentication method is a challenge and response mechanism (SCRAM-SHA-1) (page 321). Previously, MongoDB used MongoDB Challenge and Response (MONGODB-CR) (page 322) as the default.

MongoDB also supports x509 certificate authentication (page 322), LDAP proxy authentication (page 323), and Kerberos authentication (page 322).

This section introduces the mechanisms available in MongoDB.

To specify the authentication mechanism to use, see authenticationMechanisms.

SCRAM-SHA-1 Authentication

New in version 3.0.

SCRAM-SHA-1 is an IETF standard, RFC 580210, that defines best practice methods for implementation of challenge-response mechanisms for authenticating users with passwords.

SCRAM-SHA-1 verifies supplied user credentials 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.

Note: A driver upgrade is necessary to use the SCRAM-SHA-1 authentication mechanism if your current driver version does not support SCRAM-SHA-1. See required driver versions (page 819) for details.

See also:

- Blog Post: Improved Password-Based Authentication in MongoDB 3.0: SCRAM Explained (Part 1)11
- Blog Post: Improved Password-Based Authentication in MongoDB 3.0: SCRAM Explained (Part 2)12

SCRAM-SHA-1 Advantages  MongoDB’s implementation of SCRAM-SHA-1 represents an improvement in security over the previously-used MONGODB-CR, providing:

- A tunable work factor (iterationCount),
- Per-user random salts rather than server-wide salts,
- A cryptographically stronger hash function (SHA-1 rather than MD5), and
- Authentication of the server to the client as well as the client to the server.

SCRAM-SHA-1 and Existing User Credentials  SCRAM-SHA-1 is the default mechanism for MongoDB versions beginning with the 3.0 series. However, if you are upgrading a MongoDB 2.6 instances that already have users credentials, MongoDB will continue to use MONGODB-CR for challenge-response authentication until you upgrade the authentication schema. Even when using the MONGODB-CR authentication mechanism, clients and drivers that support MongoDB 3.0 features (see Driver Compatibility Changes (page 808)) will use the SCRAM communication protocol.

For details on upgrading the authentication schema model to SCRAM-SHA-1, see Upgrade to SCRAM-SHA-1 (page 818).

---

Warning: The procedure to upgrade to SCRAM-SHA-1 discards the MONGODB-CR credentials used by 2.6. As such, the procedure is irreversible, short of restoring from backups. The procedure also disables MONGODB-CR as an authentication mechanism.

MONGODB-CR Authentication

MONGODB-CR is a challenge-response mechanism that authenticates users through passwords.

Changed in version 3.0: As of version 3.0, MongoDB no longer defaults to MONGODB-CR and instead uses SCRAM-SHA-1 as the default authentication mechanism.

MONGODB-CR verifies supplied user credentials 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 serve to identify the user.

Using key files, you can also use MONGODB-CR authentication for the internal member authentication (page 323) 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 379) 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 342).

To authenticate to servers, clients can use x.509 certificates instead of usernames and passwords. See Client x.509 Certificate (page 361) 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 363) for more information.

Kerberos Authentication

MongoDB Enterprise 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 332) for MongoDB, and added Kerberos user principal (page 331) to MongoDB.

See Kerberos Authentication (page 331) for more information on Kerberos and MongoDB. To configure MongoDB to use Kerberos authentication, see Configure MongoDB with Kerberos Authentication on Linux (page 372) and Configure MongoDB with Kerberos Authentication on Windows (page 376).

LDAP Proxy Authority Authentication

MongoDB Enterprise\textsuperscript{14} supports proxy authentication through a Lightweight Directory Access Protocol (LDAP) service. See Authenticate Using SASL and LDAP with OpenLDAP (page 369) and Authenticate Using SASL and LDAP with ActiveDirectory (page 366).

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 322), x.509 (page 322), LDAP Proxy (page 323) and Kerberos (page 322) 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 378) 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 322) 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.

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 379) 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 358).

\textsuperscript{14}http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
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 669) 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 connections from the localhost interface to create the first user on the admin database. The exception applies only when there are no users created in the MongoDB instance.

Changed in version 3.0: The localhost exception changed so that these connections only have access to create the first user on the admin database. In previous versions, connections that gained access using the localhost exception had unrestricted access to 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 412) role. See Enable Client Access Control (page 357) and Create a User Administrator (page 384) for more information.

In the case of a sharded cluster, the localhost exception applies to each shard individually as well as to the cluster as a whole. Once you create a sharded cluster and add an administrator to the mongos instance, you must still prevent unauthorized access to the individual shards. Follow one of the following steps for each shard in your cluster:

- Create an administrative user, or
- Disable the localhost exception at startup. To disable the localhost exception, use setParameter in your configuration file, or --setParameter on the command line to set the enableLocalhostAuthBypass parameter to 0.

6.2.2 Authorization

MongoDB employs Role-Based Access Control (RBAC) to govern access to a MongoDB system. A user is granted one or more roles (page 325) 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 406), dbAdmin (page 406), and root (page 413) 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 defined directly in the role or inherited from another role.

A role’s privileges apply to the database where the role is created. A role created on the `admin` database can include privileges that apply to all databases or to the `cluster` (page 418).

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 412) 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 419) 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 419).

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 414).

**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 413) 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 325) that are scoped to a specific collection in a particular database, administrators can implement collection-level access control.

See Collection-Level Access Control (page 326) for more information.

Users

MongoDB stores user credentials in the protected admin.system.users (page 291). 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 416). 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 412). This user can then create all other users in the system. See Create a User Administrator (page 384).

Protect the User and Role Collections

MongoDB stores role and user data in the protected admin.system.roles (page 291) and admin.system.users (page 291) collections, which are only accessible using the user management methods.

If you disable access control, do not modify the admin.system.roles (page 291) and admin.system.users (page 291) 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 419). Also consider the reference for the form of the resource documents (page 417).

To create users see the Create a User Administrator (page 384) and Add a User to a Database (page 386) tutorials.

6.2.3 Collection-Level Access Control

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 325). By creating a role with privileges (page 325) 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 419) 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, an 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:

```json
privileges: [
    { 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 324). For a tutorial on creating user-defined roles, see Create a Role (page 388).

6.2.4 Network Exposure and Security

By default, MongoDB programs (i.e. mongos and mongod) will bind to all available network interfaces (i.e. IP addresses) on a system.

This page outlines various runtime options that allow you to limit access to MongoDB programs.

Configuration Options

You can limit the network exposure with the following mongod and mongos configuration options: 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.

**Warning:** Disable this interface for production deployments. If you enable this interface, you should only allow trusted clients to access this port. See Firewalls (page 328).
**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, 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`.

**Warning:** Disable this option for production deployments. If 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.

**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 328). See also Security Considerations (page 194) 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 335) and Configure Windows netsh Firewall for MongoDB (page 339).

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:

---

15 Starting in version 2.6, http interface is disabled by default.
For MongoDB deployments on Amazon's web services, see the Amazon EC2 page, which addresses Amazon's Security Groups and other EC2-specific security features.

**Virtual Private Networks**

Virtual private networks, or VPNs, make it possible to link two networks over an encrypted and limited-access trusted network. Typically, MongoDB users who use VPNs use 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

The following section contains strategies to limit risks related to MongoDB’s available interfaces including JavaScript, HTTP, and REST interfaces.

#### 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, 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

**Warning:** Ensure that the HTTP status interface, the REST API, and the JSON API are all disabled in production environments to prevent potential data exposure and vulnerability to attackers.

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 327).

---

16 [https://docs.mongodb.org/ecosystem/platforms/amazon-ec2](https://docs.mongodb.org/ecosystem/platforms/amazon-ec2)

17 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.`
Note: While MongoDB Enterprise does support Kerberos authentication, Kerberos is not supported in HTTP status interface in any version of MongoDB.

Changed in version 3.0.

Neither the HTTP status interface nor the REST API support the SCRAM-SHA-1 (page 321) challenge-response user authentication mechanism introduced in version 3.0.

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 335)
- Configure Windows netsh Firewall for MongoDB (page 339)

6.2.6 Auditing

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 398).

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 400) to restrict the events captured. To set up filters, see Configure Audit Filters (page 400).
Audit Guarantee

The auditing system writes every audit event 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

New in version 2.4.

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 331) and service principals (page 332).

**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>

18 Audit configuration can include a filter (page 400) to limit events to audit.
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 372) and Configure MongoDB with Kerberos Authentication on Windows (page 376).

See also:

User and Role Management Tutorials (page 384) 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/kerb5.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 378).

Linux Keytab Files

Linux systems can store Kerberos authentication keys for a service principal (page 332) 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 332).

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 Ktpass19 for more information.

Authenticate With Kerberos

To configure MongoDB for Kerberos support and authenticate, see Configure MongoDB with Kerberos Authentication on Linux (page 372) and Configure MongoDB with Kerberos Authentication on Windows (page 376).

Operational Considerations

The HTTP Console

The MongoDB HTTP Console\(^\text{20}\) 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).

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\(^\text{21}\)
- C#\(^\text{22}\)
- C++\(^\text{23}\)
- Python\(^\text{24}\)

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 372) and Configure MongoDB with Kerberos Authentication on Windows (page 376) for details.

\(^{20}\)https://docs.mongodb.org/ecosystem/tools/http-interfaces/#http-console
\(^{21}\)https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-java-driver/
\(^{22}\)https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-csharp-driver/
\(^{23}\)https://docs.mongodb.org/ecosystem/tutorial/authenticate-with-cpp-driver/
\(^{24}\)http://api.mongodb.org/python/current/examples/authentication.html
Additional Resources

- MongoDB LDAP and Kerberos Authentication with Dell (Quest) Authentication Services
- MongoDB with Red Hat Enterprise Linux Identity Management and Kerberos

6.3 Security Tutorials

The following tutorials provide instructions for enabling and using the security features available in MongoDB.

Network Security Tutorials (page 335) 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 335) Basic firewall configuration patterns and examples for iptables on Linux systems.

Configure Windows netsh Firewall for MongoDB (page 339) Basic firewall configuration patterns and examples for netsh on Windows systems.

Configure mongod and mongos for TLS/SSL (page 342) TLS/SSL allows MongoDB clients to support encrypted connections to mongod instances.

Continue reading from Network Security Tutorials (page 335) for more information on running MongoDB in secure environments.

Security Deployment Tutorials (page 353) These tutorials describe procedures for deploying MongoDB using authentication and authorization.

Access Control Tutorials (page 356) These tutorials describe procedures relevant for the configuration, operation, and maintenance of MongoDB’s access control system.

Enable Client Access Control (page 357) Describes the process for enabling authentication for MongoDB deployments.

Use x.509 Certificates to Authenticate Clients (page 361) Use x.509 for client authentication.

Use x.509 Certificate for Membership Authentication (page 363) Use x.509 for internal member authentication for replica sets and sharded clusters.

Configure MongoDB with Kerberos Authentication on Linux (page 372) For MongoDB Enterprise Linux, describes the process to enable Kerberos-based authentication for MongoDB deployments.

Continue reading from Access Control Tutorials (page 356) for additional tutorials on configuring MongoDB’s authentication systems.

Enable Authentication after Creating the User Administrator (page 359) Describes an alternative process for enabling authentication for MongoDB deployments.

User and Role Management Tutorials (page 384) 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 and setup of the authorization system.

Add a User to a Database (page 386) Create non-administrator users using MongoDB’s role-based authentication system.

Create a Role (page 388) Create custom role.

Modify a User’s Access (page 393) Modify the actions available to a user on specific database resources.

---

**View Roles** (page 394) View a role’s privileges.

Continue reading from *User and Role Management Tutorials* (page 384) for additional tutorials on managing users and privileges in MongoDB’s authorization system.

**Auditing Tutorials** (page 398) 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 403) 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 335) Basic firewall configuration patterns and examples for `iptables` on Linux systems.
- **Configure Windows `netsh` Firewall for MongoDB** (page 339) Basic firewall configuration patterns and examples for `netsh` on Windows systems.
- **Configure `mongod` and `mongos` for TLS/SSL** (page 342) TLS/SSL allows MongoDB clients to support encrypted connections to `mongod` instances.
- **TLS/SSL Configuration for Clients** (page 346) Configure clients to connect to MongoDB instances that use TLS/SSL.
- **Upgrade a Cluster to Use TLS/SSL** (page 350) Rolling upgrade process to use TLS/SSL.
- **Configure MongoDB for FIPS** (page 351) Configure for Federal Information Processing Standard (FIPS).

**Configure Linux `iptables` Firewall for MongoDB**

On contemporary Linux systems, the `iptables` program provides methods for managing the Linux Kernel’s `netfilter` or network packet filtering capabilities. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic firewall configurations for `iptables` firewalls on Linux. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see *Security Concepts* (page 320).

See also:

For MongoDB deployments on Amazon’s web services, see the Amazon EC2[^27] 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.

[^27]: https://docs.mongodb.org/ecosystem/platforms/amazon-ec2
Given the default ports (page 327) 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 338).

Patterns

This section contains a number of patterns and examples for configuring iptables for use with MongoDB deployments. If you have configured different ports using the port configuration setting, you will need to modify the rules accordingly.

Traffic to and from mongod Instances  This pattern is applicable to all mongod instances running as standalone instances or as part of a replica set.

The goal of this pattern is to explicitly allow traffic to the mongod instance from the application server. In the following examples, replace <ip-address> with the IP address of the application server:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27017 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27017 -m state --state ESTABLISHED -j ACCEPT
```

The first rule allows all incoming traffic from <ip-address> on port 27017, which allows the application server to connect to the mongod instance. The second rule, allows outgoing traffic from the mongod to reach the application server.

Optional

If you have only one application server, you can replace <ip-address> with either the IP address itself, such as: 198.51.100.55. You can also express this using CIDR notation as 198.51.100.55/32. If you want to permit a larger block of possible IP addresses you can allow traffic from a /24 using one of the following specifications for the <ip-address>, as follows:

```
10.10.10.10/24
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 336) 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. 28 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.

```
You also can run a config server by using the configsvr value for the clusterRole setting in a configuration file.

28```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT

Replace <ip-address> with the address or address space of all the mongod that provide config servers.

Additionally, config servers need to allow incoming connections from all of the mongod instances in the cluster and all mongod instances in the cluster. Add rules that resemble the following:

iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT

Replace <ip-address> with the address of the mongos instances and the shard mongod instances.

Traffic to and from a MongoDB Shard Server  For shard servers, running as mongod --shardsvr 29 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:

iptables -A INPUT -s <ip-address> -p tcp --destination-port 27018 -m state --state NEW,ESTABLISHED -j ACCEPT
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT

Replace the <ip-address> specification with the IP address of all mongod. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members, to:

- all mongod instances in the shard’s replica sets.
- all mongod instances in other shards. 30

Furthermore, shards need to be able make outgoing connections to:

- all mongos instances.
- all mongod instances in the config servers.

Create a rule that resembles the following, and replace the <ip-address> with the address of the config servers and the mongos instances:

iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27018 -m state --state ESTABLISHED -j ACCEPT

Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the --discover needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos instances.

2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

iptables -A INPUT -s <ip-address> -p tcp --destination-port 28017 -m state --state NEW,ESTABLISHED -j ACCEPT

Replace <ip-address> with the address of the instance that needs access to the HTTP or REST interface. For all deployments, you should restrict access to this port to only the monitoring instance.

Optional

For 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,ESTABLISHED -j ACCEPT

---

29 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.

30 All shards in a cluster need to be able to communicate with all other shards to facilitate chunk and balancing operations.
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,ESTABLISHED -j ACCEPT
```

Change Default Policy to DROP

The default policy for iptables chains is to allow all traffic. After completing all iptables configuration changes, you must change the default policy to DROP so that all traffic that isn’t explicitly allowed as above will not be able to reach components of the MongoDB deployment. Issue the following commands to change this policy:

```
iptables -P INPUT DROP
iptables -P OUTPUT DROP
```

Manage and Maintain iptables Configuration

This section contains a number of basic operations for managing and using iptables. There are various front end tools that automate some aspects of iptables configuration, but at the core all iptables front ends provide the same basic functionality:

**Make all iptables Rules Persistent** By default all iptables rules are only stored in memory. When your system restarts, your firewall rules will revert to their defaults. When you have tested a rule set and have guaranteed that it effectively controls traffic you can use the following operations to you should make the rule set persistent.

On Red Hat Enterprise Linux, Fedora Linux, and related distributions you can issue the following command:

```
service iptables save
```

On Debian, Ubuntu, and related distributions, you can use the following command to dump the iptables rules to the `/etc/iptables.conf` file:

```
iptables-save > /etc/iptables.conf
```

Run the following operation to restore the network rules:

```
iptables-restore < /etc/iptables.conf
```

Place this command in your `rc.local` file, or in the `/etc/network/if-up.d/iptables` file with other similar operations.

**List all iptables Rules** To list all of currently applied iptables rules, use the following operation at the system shell.

```
iptables -L
```

**Flush all iptables Rules** If you make a configuration mistake when entering iptables rules or simply need to revert to the default rule set, you can use the following operation at the system shell to flush all rules:

```
iptables -F
```

If you’ve already made your iptables rules persistent, you will need to repeat the appropriate procedure in the **Make all iptables Rules Persistent** (page 338) section.
Configure Windows `netsh` Firewall for MongoDB

On Windows Server systems, the `netsh` program provides methods for managing the Windows Firewall. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic Windows Firewall configurations. Use these approaches as a starting point for your larger networking organization. For a detailed overview of security practices and risk management for MongoDB, see Security Concepts (page 320).

See also:

Windows Firewall\(^{31}\) 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 327) 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.

```bash
netsh advfirewall firewall add rule name="Open mongod port 27017" dir=in action=allow protocol=TCP localport=27017
```

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:

```bash
netsh advfirewall firewall add rule name="Allowing mongod" dir=in action=allow program=" C:\mongodb\bin\mongod.exe"
```

You can allow all access for a mongos.exe server, with the following invocation:

```bash
netsh advfirewall firewall add rule name="Allowing mongos" dir=in action=allow program=" C:\mongodb\bin\mongos.exe"
```

**Traffic to and from mongos.exe Instances**  
mongos.exe instances provide query routing for *sharded clusters*. Clients connect to mongos.exe instances, which behave from the client’s perspective as mongod.exe instances. In turn, the mongos.exe connects to all mongod.exe instances that are components of the sharded cluster.

Use the same *Windows Firewall* command to allow traffic to and from these instances as you would from the mongod.exe instances that are members of the replica set.

```bash
netsh advfirewall firewall add rule name="Open mongod shard port 27018" dir=in action=allow protocol=TCP localport=27018
```

**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. 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.

```bash
netsh advfirewall firewall add rule name="Open mongod config svr port 27019" dir=in action=allow protocol=TCP localport=27019
```

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:

```bash
netsh advfirewall firewall add rule name="Open mongod config svr inbound" dir=in action=allow protocol=TCP remoteip=<ip-address> localport=27019
```

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* 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:

```bash
netsh advfirewall firewall add rule name="Open mongod shardsvr inbound" dir=in action=allow protocol=TCP remoteip=<ip-address> localport=27018
netsh advfirewall firewall add rule name="Open mongod shardsvr outbound" dir=out action=allow protocol=TCP remoteip=<ip-address> localport=27018
```

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.

Furthermore, shards need to be able make outgoing connections to:

- all mongos.exe instances.

---

32 You also can run a config server by using the configsvr value for the *clusterRole* setting in a configuration file.

33 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.

34 All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.
• 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 protocol=TCP remoteip=<ip-address> localport=27018

Provide Access For Monitoring Systems

1. The mongostat diagnostic tool, when running with the --discover needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the mongos.exe instances.

2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

netsh advfirewall firewall add rule name="Open mongod HTTP monitoring inbound" dir=in action=allow protocol=TCP remoteip=<ip-address> localport=28017

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=allow protocol=TCP remoteip=<ip-address> localport=28018

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 action=allow protocol=TCP remoteip=<ip-address> localport=28019

Manage and Maintain Windows Firewall Configurations

This section contains a number of basic operations for managing and using netsh. While you can use the GUI front ends to manage the Windows Firewall, all core functionality is accessible from netsh.

Delete all Windows Firewall Rules  To delete the firewall rule allowing mongod.exe traffic:

netsh advfirewall firewall delete rule name="Open mongod port 27017" protocol=tcp localport=27017

netsh advfirewall firewall delete rule name="Open mongod shard port 27018" protocol=tcp localport=27018

List All Windows Firewall Rules  To return a list of all Windows Firewall rules:

netsh advfirewall firewall show rule name=all

Reset Windows Firewall  To reset the Windows Firewall rules:

netsh advfirewall reset
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

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 350).

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.

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  New in version 3.0: Most MongoDB distributions now include support for SSL.

Certain distributions of MongoDB do not contain support for SSL. To use SSL, be sure to choose a package that supports SSL. All MongoDB Enterprise supported platforms include SSL support.

Client Support  See TLS/SSL Configuration for Clients (page 346) 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.

---


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 323), 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.key
```

This operation generates a new, self-signed certificate with no passphrase that is valid for 365 days. Once you have the certificate, concatenate the certificate and private key to a .pem file, as in the following example:

```
cat mongodb-cert.key mongodb-cert.crt > mongodb.pem
```

See also:
Use x.509 Certificates to Authenticate Clients (page 361)

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 345).

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 configuration file format:\^\[37\]:

    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 346) for more information on connecting to mongod and mongos running with SSL.

See also:
Upgrade a Cluster to Use TLS/SSL (page 350)

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 345).

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 configuration file format\^\[38\]:

```
37http://docs.mongodb.org/v2.4/reference/configuration-options
38http://docs.mongodb.org/v2.4/reference/configuration-options
```

[^37]: http://docs.mongodb.org/v2.4/reference/configuration-options
[^38]: 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 346) for more information on connecting to mongod and mongos running with SSL.

See also:
Upgrade a Cluster to Use TLS/SSL (page 350)

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 --sslCAFile /etc/ssl/ca.pem
```

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 allowConnectionsWithoutCertificates 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 allowConnectionsWithoutCertificates setting:

```
mongod --sslMode requireSSL --sslAllowConnectionsWithoutCertificates --sslPEMKeyFile /etc/ssl/mongodb.pem --sslCAFile /etc/ssl/ca.pem
```

Then, clients can connect either with the option --ssl and no certificate or with the option --ssl and a valid certificate. See TLS/SSL Configuration for Clients (page 346) 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

Note: FIPS-compatible SSL is available only in MongoDB Enterprise. See Configure MongoDB for FIPS (page 351) for more information.

See Configure MongoDB for FIPS (page 351) for more details.

TLS/SSL Configuration for Clients

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 342).

mongo Shell SSL Configuration

For SSL connections, you must use the mongo shell built with SSL support or distributed with MongoDB Enterprise. New in version 3.0: Most MongoDB distributions now include support for SSL.

The mongo shell provides various `mongo-shell-ssl` settings, including:

- `--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).

Changed in version 3.0: When running mongo with the `--ssl` option, you must include either `--sslCAFile` or `--sslAllowInvalidCertificates`.

This restriction does not apply to the MongoDB tools. However, running the tools without `--sslCAFile` creates the same vulnerability to invalid certificates.

Warning: For SSL connections (`--ssl`) to mongod and mongos, if the mongo shell (or MongoDB tools (page 350)) runs without the `--sslCAFile <CAFile>` option (i.e. specifies the `--sslAllowInvalidCertificates` instead), the mongo shell (or MongoDB tools (page 350)) will not attempt to validate the server certificates. This creates a 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 to validate the server certificates in cases where intrusion is a possibility.

---

Connect to MongoDB Instance with SSL Encryption

To connect to a mongod or mongos instance that requires only a SSL encryption mode (page 343), start mongo shell with --ssl and include the --sslCAFile to validate the server certificates.

```
mongo --ssl --sslCAFile /etc/ssl/ca.pem
```

Changed in version 3.0: When running mongo with the --ssl option, you must include either --sslCAFile or --sslAllowInvalidCertificates.

This restriction does not apply to the MongoDB tools. However, running the tools without --sslCAFile creates the same vulnerability to invalid certificates.

Connect to MongoDB Instance that Requires Client Certificates

To connect to a mongod or mongos that requires CA-signed client certificates (page 344), start the mongo shell with --ssl, the --sslPEMKeyFile option to specify the signed certificate-key file, and the --sslCAFile to validate the server certificates.

```
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem --sslCAFile /etc/ssl/ca.pem
```

Changed in version 3.0: When running mongo with the --ssl option, you must include either --sslCAFile or --sslAllowInvalidCertificates.

This restriction does not apply to the MongoDB tools. However, running the tools without --sslCAFile creates the same vulnerability to invalid certificates.

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 345), start mongo shell either:

- with the --ssl, --sslCAFile, and no certificate or
- with the --ssl, --sslCAFile, and a valid signed certificate.

Changed in version 3.0: When running mongo with the --ssl option, you must include either --sslCAFile or --sslAllowInvalidCertificates.

This restriction does not apply to the MongoDB tools. However, running the tools without --sslCAFile creates the same vulnerability to invalid certificates.

For example, if mongod is running with weak certificate validation, both of the following mongo shell clients can connect to that mongod:

```
mongo --ssl --sslCAFile /etc/ssl/ca.pem
mongo --ssl --sslPEMKeyFile /etc/ssl/client.pem --sslCAFile /etc/ssl/ca.pem
```

**Important:** If the client presents a certificate, the certificate must be valid.

**MongoDB Cloud Manager and Ops Manager Monitoring Agent**

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\(^{40}\) for more information about SSL configuration. For Ops Manager, see Ops Manager documentation\(^{41}\).

**PyMongo**

Add the “ssl=True” parameter to a PyMongo `MongoClient`\(^{42}\) to create a MongoDB connection to an SSL MongoDB instance:

```python
from pymongo import MongoClient
c = MongoClient(host="mongodb.example.net", port=27017, ssl=True)
```

To connect to a replica set, use the following operation:

```python
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\(^{43}\).

**Java**

Consider the following example “SSLApp.java” class file:

```java
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" );
        DCollections c = db.getCollection( "foo" );

        System.out.println( c.findOne() );
    }
}
```

For more details, see the Java MongoDB Driver page\(^{44}\).

\(^{40}\)https://docs.cloud.mongodb.com/
\(^{41}\)https://docs.opsmanager.mongodb.com/current
\(^{42}\)http://api.mongodb.org/python/current/api/pymongo/mongo_client.html#pymongo.mongo_client.MongoClient
\(^{43}\)https://docs.mongodb.org/ecosystem/drivers/python
\(^{44}\)https://docs.mongodb.org/ecosystem/drivers/java
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:

```ruby
gem install mongo
```

Then connect to a standalone instance, using the following form:

```ruby
require 'rubygems'
require 'mongo'

connection = MongoClient.new('localhost', 27017, :ssl => true)
```

Replace `connection` with the following if you're connecting to a replica set:

```ruby
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\(^{45}\).

Node.JS (node-mongodb-native)

In the `node-mongodb-native`\(^{46}\) driver, use the following invocation to connect to a `mongod` or `mongos` instance via SSL:

```javascript
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:

```javascript
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\(^{47}\).

.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:

```csharp
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

---
\(^{45}\) [https://docs.mongodb.org/ecosystem/drivers/ruby](https://docs.mongodb.org/ecosystem/drivers/ruby)

\(^{46}\) [https://github.com/mongodb/node-mongodb-native](https://github.com/mongodb/node-mongodb-native)

\(^{47}\) [https://docs.mongodb.org/ecosystem/drivers/node-js](https://docs.mongodb.org/ecosystem/drivers/node-js)
you encounter this issue, add the `sslverifycertificate=false` option to the connection string to prevent the .NET driver from validating the certificate, as follows:

```csharp
var connectionString = "mongodb://localhost/?ssl=true&sslverifycertificate=false";
var server = MongoServer.Create(connectionString);
```

For more details, see the .NET MongoDB Driver page.

**MongoDB Tools**

Changed in version 2.6.

Various MongoDB utility programs support SSL. These tools include:

- mongodump
- mongoexport
- mongofiles
- mongoinport
- 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 346).

**Upgrade a Cluster to Use TLS/SSL**

Changed in version 3.0: Most MongoDB distributions now include support for TLS/SSL. See *Configure mongod and mongos for TLS/SSL* (page 342) and *TLS/SSL Configuration for Clients* (page 346) 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 342) as well as any other options that are required for your specific configuration. For example:

   ```csharp
   mongod --replSet <name> --sslMode allowSSL --sslPEMKeyFile <path to TLS/SSL Certificate and key>
   ```

48https://docs.mongodb.org/ecosystem/drivers/csharp
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:

```yaml
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:\footnote{http://docs.mongodb.org/v2.4/reference/configuration-options}

```plaintext
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 \textit{TLS/SSL Configuration for Clients} (page 346).

3. For each node of a cluster, use the \texttt{setParameter} command to update the \texttt{sslmode} to \texttt{preferSSL}.\footnote{As an alternative to using the \texttt{setParameter} command, you can also restart the nodes with the appropriate TLS/SSL options and values.} With \texttt{preferSSL} as its \texttt{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:

```javascript
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 \texttt{setParameter} command to update the \texttt{sslmode} to \texttt{requireSSL}. With \texttt{requireSSL} as its \texttt{net.ssl.mode}, the node will reject any non-TLS/non-SSL connections. For example:

```javascript
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.

\section*{Configure MongoDB for FIPS}

New in version 2.6.

\subsection*{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.

\subsection*{Prerequisites}

\textbf{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\(^{51}\) version supports FIPS mode. See Install MongoDB Enterprise (page 33) to download and install MongoDB Enterprise\(^{52}\) 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"
```

**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 must 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 342) 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 342).

**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
```

\(^{51}\)http://www.mongodb.com/products/mongodb-enterprise?jmp=docs

\(^{52}\)http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
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 353) Configure a replica set that has authentication enabled.

Deploy Replica Set and Configure Authentication and Authorization

Overview

With authentication (page 320) enabled, MongoDB forces all clients to identify themselves before granting access to the server. Authorization (page 324), 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 559) 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 323). 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 572) 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 system’s `/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](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 413) user (`siteRootAdmin`) that you will use to complete the remainder of the tutorial:

```javascript
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:

```bash
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 652):

```
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`. 

6.3. Security Tutorials
Step 11: Check the status of the replica set. Use the `rs.status()` operation:

```javascript
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 406) role to create a user who can update data, or the `read` (page 407) role to create a user who can search data but no more. You also can define custom roles (page 325).

For example, the following creates a database administrator for the `products` database:

```javascript
use products
db.createUser(
    {
        user: "productsDBAdmin",
        pwd: "password",
        roles: [
            {
                role: "dbOwner",
                db: "products"
            }
        ]
    }
)
```

For an overview of roles and privileges, see Authorization (page 324). For more information on adding users, see Add a User to a Database (page 386).

### 6.3.3 Access Control Tutorials

The following tutorials provide instructions for MongoDB’s authentication and authorization related features.

Enable Client Access Control (page 357) Describes the process for enabling authentication for MongoDB deployments.

Enable Authentication in a Sharded Cluster (page 358) 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 359) Describes an alternative process for enabling authentication for MongoDB deployments.

Use x.509 Certificates to Authenticate Clients (page 361) Use x.509 for client authentication.

Use x.509 Certificate for Membership Authentication (page 363) Use x.509 for internal member authentication for replica sets and sharded clusters.

Authenticate Using SASL and LDAP with ActiveDirectory (page 366) Describes the process for authentication using SASL/LDAP with ActiveDirectory.

Authenticate Using SASL and LDAP with OpenLDAP (page 369) Describes the process for authentication using SASL/LDAP with OpenLDAP.

Configure MongoDB with Kerberos Authentication on Linux (page 372) For MongoDB Enterprise Linux, describes the process to enable Kerberos-based authentication for MongoDB deployments.

Configure MongoDB with Kerberos Authentication on Windows (page 376) For MongoDB Enterprise for Windows, describes the process to enable Kerberos-based authentication for MongoDB deployments.
Authenticate to a MongoDB Instance or Cluster (page 378)  Describes the process for authenticating to MongoDB systems using the mongo shell.

Generate a Key File (page 379)  Use key file to allow the components of MongoDB sharded cluster or replica set to mutually authenticate.

Troubleshoot Kerberos Authentication on Linux (page 380)  Steps to troubleshoot Kerberos-based authentication for MongoDB deployments.

Implement Field Level Redaction (page 381)  Describes the process to set up and access document content that can have different access levels for the same data.

Enable Client Access Control

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 324). In that case, you must use the “Enable Authentication after Creating the User Administrator (page 359)” procedure to enable access control.

This procedure uses the localhost exception (page 324) to allow you to create the first user after enabling authentication. See Localhost Exception (page 324) and Authentication (page 320) 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 324).
Step 3: Create the system user administrator. Add the user with the userAdminAnyDatabase role, and only that role.

The following example creates the user siteUserAdmin user on the admin database:

```javascript
use admin
db.createUser(
  
  {
    user: "siteUserAdmin",
    pwd: "password",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

After you create the user administrator, the localhost exception (page 324) 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 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 386).

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

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 key file. The content of the key file must be the same on all cluster members.

Considerations

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.

As described in Localhost Exception (page 324), the localhost exception will apply to the individual shards unless you either create an administrative user or disable the localhost exception on each shard.
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 384).

Related Documents

- **Authentication** (page 320)
- **Security** (page 317)
- **Use x.509 Certificate for Membership Authentication** (page 363)

Enable Authentication after Creating the User Administrator

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 357).
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 324) to gain access to a system with no users and authentication enabled. See Enable Client Access Control (page 357) 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 224) or Deploy a Sharded Cluster (page 691).

Step 2: Create the system user administrator. Add the user with the userAdminAnyDatabase (page 412) 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 386).

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

New in version 2.6.

MongoDB supports x.509 certificate authentication for use with a secure TLS/SSL connection (page 342). The x.509 client authentication allows clients to authenticate to servers with certificates (page 361) 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 363).

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 363). 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 363), 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 PEM file>
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\(^{53}\), 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:

   ```
   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 }
   }
   ```

\(^{53}\)http://docs.mongodb.org/v2.4/reference/configuration-options
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 386) 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 362).

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 362) for the `user` field.

For example, if using the `mongo` shell,

1. Connect `mongo` shell to the `mongod` set up for SSL:

   ```
   mongo --ssl --sslPEMKeyFile <path to CA signed client PEM file> --sslCAFile <path to root CA PEM file>
   ```

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**

New in version 2.6.

MongoDB supports x.509 certificate authentication for use with a secure *TLS/SSL connection* (page 342). 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 320). The membership authentication is an internal process.

For client authentication with x.509, see *Use x.509 Certificates to Authenticate Clients* (page 361).

**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.

\[
\begin{align*}
CN=host1,OU=Dept1,O=MongoDB,ST=NY,C=US \\
C=US, ST=CA, O=MongoDB, OU=Dept1, CN=host2
\end{align*}
\]

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.

\[
\begin{align*}
CN=host1,OU=Dept1,OU=Sales,O=MongoDB \\
CN=host2,OU=Dept1,O=MongoDB
\end{align*}
\]

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:

\[
\begin{align*}
subject= CN=<myhostname1>,OU=Dept1,O=MongoDB,ST=NY,C=US \\
subject= CN=<myhostname2>,OU=Dept1,O=MongoDB,ST=NY,C=US \\
subject= CN=<myhostname3>,OU=Dept1,O=MongoDB,ST=NY,C=US
\end{align*}
\]

You can use an x509 certificate that does not have Extended Key Usage (EUKU) 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 membership certificate and key PEM file> --sslPEMKeyFile <path to SSL certificate and key PEM file> --sslCAFile <path to root CA PEM file>
```

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 \textit{SSL Certificate Passphrase} (page 345) for details.

\textbf{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:

```yaml
security:
  clusterAuthMode: x509
net:
  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 342) as well as any other options that are required for your specific configuration. For example:

   ```
   mongod --replSet <name> --sslMode requireSSL --clusterAuthMode sendKeyFile --sslClusterFile <path to membership certificate and key PEM file> --sslPEMKeyFile <path to SSL Certificate and key PEM file> --sslCAFile <path to root CA PEM file>
   ```

   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`. For example,

   ```
   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. 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.

---

As an alternative to using the `setParameter` command, you can also restart the nodes with the appropriate TLS/SSL and x509 options and values.
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 342) 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. For example:

   db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "preferSSL", clusterAuthMode: "sendX509" } )

   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. For example:

   db.getSiblingDB('admin').runCommand( { setParameter: 1, sslMode: "requireSSL", clusterAuthMode: "x509" } )

   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

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 866) 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 if 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.

**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\(^{56}\) of saslauthd.

To use with ActiveDirectory, start saslauthd with the following configuration options set in the saslauthd.conf file:

```
ladp_servers: <ldap uri>
ladp_use_sasl: yes
ladp_mech: DIGEST-MD5
ladp_auth_method: fastbind
```

For the `<ldap uri>`, specify the uri of the ldap server. For example, `ladp_servers: ldaps://ad.example.net`.

For more information on saslauthd configuration, see [http://www.openldap.org/doc/admin24/guide.html#Configuringsaslauthd](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
```

---

\(^{55}\)[http://www.linuxcommand.org/man_pages/saslauthd8.html]

\(^{56}\)[http://www.linuxcommand.org/man_pages/saslauthd8.html]
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 324) to the user.

For example, the following adds a user with read-only access to the records database.

```javascript
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.

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 authenticationMechanisms=PLAIN
```

Or if using a YAML format configuration file, specify the following settings in the file:

```yaml
security:
  authorization: enabled

setParameter:
  saslauthdPath: /<some>/<path>/saslauthd/mux
  authenticationMechanisms: PLAIN
```

Or, if using the older configuration file format\(^57\):

```yaml
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:

```
```

\(^57\)http://docs.mongodb.org/v2.4/reference/configuration-options
security:
  authorization: enabled

setParameter:
  saslauthdPath:"
  authenticationMechanisms: PLAIN

Or, if using the older configuration file format58:

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
  }
)
```

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

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 866) 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.

58http://docs.mongodb.org/v2.4/reference/configuration-options

6.3. Security Tutorials
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 of saslauthd.

**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 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)
```

---

59http://www.linuxcommand.org/man_pages/saslauthd8.html
60http://www.linuxcommand.org/man_pages/saslauthd8.html
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 324) to the user.

For example, the following adds a user with read-only access to the records database.

```javascript
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.

**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 authenticationMechanisms=PLAIN
```

Or if using a YAML format configuration file, specify the following settings in the file:

```yaml
security:
  authorization: enabled

setParameter:
  saslauthdPath: /<some>/<path>/saslauthd/mux
  authenticationMechanisms: PLAIN
```

Or, if using the older configuration file format:

```
61http://docs.mongodb.org/v2.4/reference/configuration-options
```

63 Security Tutorials
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:

```
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
  }
)
```

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

New in version 2.4.

Overview

MongoDB Enterprise supports authentication using a [Kerberos service](page 331). Kerberos is an industry standard authentication protocol for large client/server system.

---

\[^{62}\]http://docs.mongodb.org/v2.4/reference/configuration-options
Prerequisites

Setting up and configuring a Kerberos deployment is beyond the scope of this document. This tutorial assumes you have configured a Kerberos service principal (page 332) for each mongod and mongos instance in your MongoDB deployment, and you have a valid keytab file (page 332) for for each mongod and mongos instance.

To verify MongoDB Enterprise binaries:

```bash
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 324) to the user.

The following example adds the Kerberos principal `application/reporting@EXAMPLE.NET` with read-only access to the `records` database:

```javascript
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:

---

6.3. Security Tutorials 373
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 332) may differ. Modify or include additional mongod options as required for your configuration. The keytab file (page 332) 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 374) 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 KRB5_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 332) 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:

```bash
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 332) may differ. The keytab file (page 332) 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 363) 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:

```yaml
setParameter:
  authenticationMechanisms: GSSAPI
```

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

```ini
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:

```yaml
security:
  authorization: enabled
setParameter:
  authenticationMechanisms: GSSAPI
storage:
  dbPath: /opt/mongodb/data
```

Or, if using the older configuration file format:\

```ini
auth = true
setParameter=authenticationMechanisms=GSSAPI
dbpath=/opt/mongodb/data
```

To start mongod with Kerberos support, use the following form:

```bash
env KRB5_KTNAME=/opt/mongodb/mongod.keytab \
/opt/mongodb/bin/mongod --config /opt/mongodb/mongod.conf
```

The path to your mongod, keytab file (page 332), and configuration file may differ. The keytab file (page 332) 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 380).

---

http://docs.mongodb.org/v2.4/reference/configuration-options
Incorporate Additional Authentication Mechanisms  Kerberos authentication *(GSSAPI (page 322) (Kerberos))* can work alongside MongoDB’s challenge/response authentication mechanisms *(SCRAM-SHA-1 (page 321) and MONGODB-CR (page 322), MongoDB’s authentication mechanism for LDAP *(PLAIN (page 323) (LDAP SASL)), and MongoDB’s authentication mechanism for x.509 ( MONGODB-X509 (page 322))*). Specify the mechanisms as follows:

```
--setParameter authenticationMechanisms=GSSAPI,SCRAM-SHA-1
```

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[^64]
- MongoDB with Red Hat Enterprise Linux Identity Management and Kerberos[^65]

Configure MongoDB with Kerberos Authentication on Windows

New in version 2.6.

Overview

MongoDB Enterprise supports authentication using a *Kerberos service* (page 331). 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 332) 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 `mongo.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 324) to the user.

The following example adds the Kerberos principal `reportingapp@EXAMPLE.NET` with read-only access to the `records` database:

```javascript
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 378).

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.

```javascript
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 378).

`mongos.exe --setParameter authenticationMechanisms=GSSAPI <additional mongos options>`

For example, the following starts a `mongos` instance with Kerberos support:
mongo.exe --setParameter authenticationMechanisms=GSSAPI --configdb shard0.example.net, shard1.example.net, shard2.example.net --keyFile C:\mongos.keyfile

Modify or include any additional mongo.exe 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 363) 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** (page 322) (Kerberos)) can work alongside MongoDB’s challenge/response authentication mechanisms (**SCRAM-SHA-1** (page 321) and **MONGODB-CR** (page 322)), MongoDB’s authentication mechanism for LDAP (**PLAIN** (page 323) (LDAP SASL)), and MongoDB’s authentication mechanism for x.509 (**MONGODB-X509** (page 322)). Specify the mechanisms as follows:

`--setParameter authenticationMechanisms=GSSAPI,SCRAM-SHA-1`

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**

**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

**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:

```
ec bo -- "my secret key" > key1
ec bo -- "my secret key\n" > key2
ec bo -- "my secret key" > key3
ec bo -- "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

New in version 2.4.

Kerberos Configuration Checklist

If you have difficulty starting `mongod` or `mongos` with Kerberos (page 331) on Linux systems, ensure that:

- The `mongod` and `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. MongoDB Enterprise does not support Kerberos authentication over the HTTP Console interface.

- Either the service principal name (SPN) in the `keytab file` (page 332) 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.

- 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.

---

66https://docs.mongodb.org/ecosystem/tools/http-interface/#http-console
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:

```bash
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/mongodb.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 380), in particular, whether the SPN in the keytab file (page 332) matches the SPN for the mongod or mongos instance.

To determine whether the SPNs match:

1. Examine the keytab file, with the following command:
   ```bash
   klist -k <keytab>
   
   Replace <keytab> with the path to your keytab file.
   ```

2. Check the configured hostname for your system, with the following command:
   ```bash
   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 331)
- Configure MongoDB with Kerberos Authentication on Linux (page 372)
- Configure MongoDB with Kerberos Authentication on Windows (page 376)

Implement Field Level Redaction

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:

```json
{  
    _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" ] ],
```
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:

```javascript
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:

```javascript
{ "_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",
```
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 384) 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 386) Create non-administrator users using MongoDB’s role-based authentication system.

*Create an Administrative User with Unrestricted Access* (page 387) 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 388) Create custom role.

*Assign a User a Role* (page 390) 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 391) View a user’s current privileges.

*Modify a User’s Access* (page 393) Modify the actions available to a user on specific database resources.

*View Roles* (page 394) View a role’s privileges.

*Change a User’s Password* (page 395) 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 396) Users with sufficient access can change their own passwords and modify the optional *custom data* associated with their user credential.

**Create a User Administrator**

**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 412) roles, which grant access to *actions* (page 419) that support user and role management. Following the policy of *least privilege* `userAdmin` (page 407) and `userAdminAnyDatabase` (page 412) 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 412) can grant itself any privilege *in the system.*
Prerequisites

Required Access You must have the createUser (page 420) action (page 419) on a database to create a new user on that database.

You must have the grantRole (page 420) action (page 419) on a role’s database to grant the role to another user.
If you have the userAdmin (page 407) or userAdminAnyDatabase (page 412) 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 324) 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 324) 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 385).

```
mongo --port 27017 -u manager -p 123456 --authenticationDatabase admin
```

Step 2: Create the system user administrator. Add the user with the userAdminAnyDatabase (page 412) 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" } ]
    }
)
```
Add a User to a Database

Changed in version 2.6.

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 325). 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 317).

Considerations

For users that authenticate using external mechanisms,⁷⁰ you do not need to provide credentials when creating users. For all users, select the roles that have the exact required privileges (page 325). If the correct roles do not exist, create roles (page 388).

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 320), 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 384).

---

⁷⁰ Configure MongoDB with Kerberos Authentication on Linux (page 372), Authenticate Using SASL and LDAP with OpenLDAP (page 369), Authenticate Using SASL and LDAP with ActiveDirectory (page 366), and x.509 certificates provide external authentication mechanisms.
**Required Access**  You must have the `createUser` (page 420) action (page 419) on a database to create a new user on that database.

You must have the `grantRole` (page 420) action (page 419) on a role’s database to grant the role to another user.

If you have the `userAdmin` (page 407) or `userAdminAnyDatabase` (page 412) 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 324) 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 386) section.

The following procedure uses the `siteUserAdmin` created in Create a User Administrator (page 384).

```
mongo --port 27017 -u siteUserAdmin -p password --authenticationDatabase admin
```

**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**

**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 419).

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.

6.3. Security Tutorials 387
If users truly need unrestricted access to a MongoDB deployment, MongoDB provides a built-in role (page 405) named root (page 413) that grants the combined privileges of all built-in roles. This document describes how to create an administrative user with the root (page 413) 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 419) on a database to create a new user on that database.

You must have the grantRole (page 420) action (page 419) on a role’s database to grant the role to another user.

If you have the userAdmin (page 407) or userAdminAnyDatabase (page 412) 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 324) or use the --noauth option when starting mongod to gain full access to 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 388) section.

The following procedure uses the siteUserAdmin created in Create a User Administrator (page 384).

```
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 413) 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

**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 419) on that database resource (page 418).
- the grantRole (page 420) action (page 419) 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 412) 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()`.

Step 1: Connect to MongoDB with the appropriate privileges. Connect to `mongod` or `mongos` with the privileges specified in the Prerequisites (page 389) section.

The following procedure uses the `siteUserAdmin` created in Create a User Administrator (page 384).

```
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 built-in role `clusterMonitor` (page 409) also provides the privilege to run `db.currentOp()` along with other privileges, and the built-in role `hostManager` (page 410) provides the privilege to run `db.killOp()` along with other privileges.
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`.

**Step 1: Connect to MongoDB with the appropriate privileges.** Connect to mongod or mongos with the privileges specified in the *Prerequisites* (page 389) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 384).

```
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**

Changed in version 2.6.

**Overview**

A role provides a user privileges to perform a set of *actions* (page 419) 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 324). For descriptions of the access each built-in role provides, see the section on *built-in roles* (page 405).

---

72 The built-in role `clusterMonitor` (page 409) also provides the privilege to run `mongostat` along with other privileges.
**Prerequisites**

You must have the `grantRole` (page 420) `action` (page 419) 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 419) 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 391) section.

The following procedure uses the `siteUserAdmin` created in `Create a User Administrator` (page 384).

```
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 386), 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 388) 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 386).

```
use reporting
db.grantRolesToUser(
   "reportsUser",
   [
    { role: "readWrite", db: "products" } ,
    { role: "readAnyDatabase", db:"admin" }
   ]
)
```

**Verify User Privileges**

**Overview**

A user’s privileges determine the access the user has to MongoDB `resources` (page 417) and the `actions` (page 419) 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 324).

**Prerequisites**

To view a role’s information, you must be explicitly granted the role or must have the *viewRole* (page 420) action (page 419) 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 384).

```
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 386), 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.js" }
  }
]  
```
Modify a User’s Access

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 324). 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 419) on a database to grant a role on that database.

You must have the `revokeRole` (page 420) action (page 419) 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 419) 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 384).

```
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 386), issue:

```
use reporting
db.getUser("reportsUser")
```
To display the privileges granted to the user by the `readWrite` role on the "accounts" database, issue:

```javascript
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 388) 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 388) 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 406) role on the `accounts` database from the `reportsUser`:

```javascript
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:

```javascript
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**

**Overview**

A `role` (page 325) 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 291) 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 419) 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:

```javascript
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:

```javascript
{ role: "<role name>", db: "<role db>" }
```

To view the custom appWriter role in the orders database, issue the following command from the mongo shell:

```javascript
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:

```javascript
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:

```javascript
db = db.getSiblingDB('admin')
db.system.roles.find()
```

Change a User’s Password

Changed in version 2.6.

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:

```bash
openssl rand -base64 48
```

Prerequisites

You must have the changeAnyPassword action (page 419) 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 419) on your database. See *Change Your Password and Custom Data* (page 396).

**Procedure**

**Step 1: Connect to MongoDB with the appropriate privileges.** Connect to the `mongod` or `mongos` with the privileges specified in the *Prerequisites* (page 395) section.

The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 384).  

```
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 `SOh3tYhxulW8ypJxmt1oOfL`:

```
db.changeUserPassword("reporting", "SOh3tYhxulW8ypJxmt1oOfL")
```

*Change Your Password and Custom Data*

Changed in version 2.6.

**Overview**

Users with appropriate privileges can change their own passwords and custom data. *Custom data* (page 417) 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 419) 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 412) role. The following procedure uses the `siteUserAdmin` created in *Create a User Administrator* (page 384).

```
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).

```javascript
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 406) and the user-created "changeOwnPasswordCustomDataRole".

```javascript
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 396) section.

```bash
mongo --port 27017 -u user123 -p 12345678 --authenticationDatabase test
```

To check that you have the privileges specified in the *Prerequisites* (page 396) 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 the user’s password to `KNlZmiaNUp0B` and custom data to `{ title: "Senior Manager" }`:

```javascript
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 398) Enable and configure MongoDB Enterprise system event auditing feature.

Configure Audit Filters (page 400) Specify which events to audit.

Configure System Events Auditing

New in version 2.6.

MongoDB Enterprise supports auditing (page 330) 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:

```bash
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:

```yaml
storage:
  dbPath: data/db
auditLog:
  destination: syslog
```

73https://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:
Configure Audit Filters

MongoDB Enterprise\textsuperscript{74} supports auditing (page 330) of various operations. When enabled (page 398), 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:

\[
\{ \text{<field1>}: \text{<expression1>}, \ldots \}\]

- The \text{<field>} can be any field in the audit message (page 424), including fields returned in the param (page 425) document.
- The \text{<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:

\[
\{ \text{atype}: \{ \$in: [ "createCollection", "dropCollection" ] \} \}
\]

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

\[
\text{mongod --dbpath data/db --auditDestination file --auditFilter '\{ \text{atype}: \{ \$in: [ "createCollection", "dropCollection" ] \} \}'}
\]

To specify the audit filter in a configuration file, you must use the YAML format of the configuration file.

\[
\text{storage:
  dbPath: data/db
  auditLog:
    destination: file
    format: BSON
    path: data/db/auditLog.bson
    filter: '\{ \text{atype}: \{ \$in: [ "createCollection", "dropCollection" ] \} \}'}
\]

Filter on Authentication Operations on a Single Database The \text{<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:

\footnote{https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs}
To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```bash
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ atype: "authenticate", "param.db": "test" }'
```

To specify the audit filter in a configuration file, you must use the YAML format of the configuration file.

```yaml
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:

```yaml
{ 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.

```bash
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ atype: { $in: [ "createCollection", "dropCollection" ] }, "param.ns": /^test\/. } }'
```

To specify the audit filter in a configuration file, you must use the YAML format of the configuration file.

```yaml
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 406) role on the test database, including users with roles that inherit from `readWrite` (page 406), by using the filter:

```yaml
{ roles: { role: "readWrite", db: "test" } }
```

To specify an audit filter, enclose the filter document in single quotes to pass the document as a string.

```bash
mongod --dbpath data/db --auth --auditDestination file --auditFilter '{ roles: { role: "readWrite", db: "test" } }'
```

To specify the audit filter in a configuration file, you must use the YAML format of the configuration file.
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.  

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:

```json
{ 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.

```bash
```

To specify the audit filter in a configuration file, you must use the YAML format of the configuration file.

```yaml
storage:
  dbPath: data/db
security:
  authorization: enabled
auditLog:
  destination: file
  format: BSON
  path: data/db/auditLog.bson
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.

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 for the collection orders in the database test by using the filter:

```json
{ atype: "authCheck", "param.ns": "test.orders", "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.

---

75 You can enable auditAuthorizationSuccess parameter without enabling --auth; however, all operations will return success for authorization checks.
mongod --dbpath data/db --auth --setParameter auditAuthorizationSuccess=true --auditDestination file --auditFilter '{ atype: "authCheck", "param.ns": "test.orders", "param.command": { $in: [ "find", "insert", "delete", "update", "findandmodify" ] } }' --auditFormat BSON --auditPath data/db/auditLog.bson

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", "insert", "delete", "update", "findandmodify" ] } })'

setParameter: { auditAuthorizationSuccess: true }

See also:

Configure System Events Auditing (page 398), Auditing (page 330), System Event Audit Messages (page 424)

6.3.6 Create a Vulnerability Report

If you believe you have discovered a vulnerability in MongoDB or have experienced a security incident related to MongoDB, please report the issue to aid in its resolution.

To report an issue, we strongly suggest filing a ticket in the SECURITY project in JIRA. MongoDB, Inc responds to vulnerability notifications within 48 hours.

Create the Report in JIRA

Submit a Ticket in the Security project on our JIRA. The ticket number will become the reference identification for the issue for its lifetime. You can use this identifier for tracking purposes.

Information to Provide

All vulnerability reports should contain as much information as possible so MongoDB’s developers can move quickly to resolve the issue. In particular, please include the following:

• The name of the product.
• Common Vulnerability information, if applicable, including:
  • CVSS (Common Vulnerability Scoring System) Score.
  • CVE (Common Vulnerability and Exposures) Identifier.
• Contact information, including an email address and/or phone number, if applicable.

Send the Report via Email

While JIRA is the preferred reporting method, you may also report vulnerabilities via email to security@mongodb.com.

6.3. Security Tutorials 403
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[80] 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

#### 6.4.1 Security Methods in the mongo Shell

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.auth()</td>
<td>Authenticated a user to a database.</td>
</tr>
</tbody>
</table>

**User Management Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.createUser()</td>
<td>Creates a new user.</td>
</tr>
<tr>
<td>db.updateUser()</td>
<td>Updates user data.</td>
</tr>
<tr>
<td>db.changeUserPassword()</td>
<td>Changes an existing user’s password.</td>
</tr>
<tr>
<td>db.removeUser()</td>
<td>Deprecated. Removes a user from a database.</td>
</tr>
<tr>
<td>db.dropAllUsers()</td>
<td>Deletes all users associated with a database.</td>
</tr>
<tr>
<td>db.dropUser()</td>
<td>Removes a single user.</td>
</tr>
<tr>
<td>db.grantRolesToUser()</td>
<td>Grants a role and its privileges to a user.</td>
</tr>
<tr>
<td>db.revokeRolesFromUser()</td>
<td>Removes a role from a user.</td>
</tr>
<tr>
<td>db.getUser()</td>
<td>Returns information about the specified user.</td>
</tr>
<tr>
<td>db.getUsers()</td>
<td>Returns information about all users associated with a database.</td>
</tr>
</tbody>
</table>

[80]https://jira.mongodb.org/browse/SECURITY
Role Management Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.createRole()</td>
<td>Creates a role and specifies its privileges.</td>
</tr>
<tr>
<td>db.updateRole()</td>
<td>Updates a user-defined role.</td>
</tr>
<tr>
<td>db.dropRole()</td>
<td>Deletes a user-defined role.</td>
</tr>
<tr>
<td>db.dropAllRoles()</td>
<td>Deletes all user-defined roles associated with a database.</td>
</tr>
<tr>
<td>db.grantPrivilegesToRole()</td>
<td>Assigns privileges to a user-defined role.</td>
</tr>
<tr>
<td>db.revokePrivilegesFromRole()</td>
<td>Removes the specified privileges from a user-defined role.</td>
</tr>
<tr>
<td>db.grantRolesToRole()</td>
<td>Specifies roles from which a user-defined role inherits privileges.</td>
</tr>
<tr>
<td>db.revokeRolesFromRole()</td>
<td>Removes a role from a user.</td>
</tr>
<tr>
<td>db.getRole()</td>
<td>Returns information for the specified role.</td>
</tr>
<tr>
<td>db.getRoles()</td>
<td>Returns information for all the user-defined roles in a database.</td>
</tr>
</tbody>
</table>

6.4.2 Security Reference Documentation

**Built-In Roles** (page 405) Reference on MongoDB provided roles and corresponding access.

**system.roles Collection** (page 413) Describes the content of the collection that stores user-defined roles.

**system.users Collection** (page 416) Describes the content of the collection that stores users’ credentials and role assignments.


**Privilege Actions** (page 419) 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**

MongoDB grants access to data and commands through *role-based authorization* (page 325) 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 325).

A role grants privileges to perform sets of *actions* (page 419) 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 291).

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 291), *system.js* (page 291), and *system.namespaces* (page 291) collections. The role provides read access by granting the following *actions* (page 419):

- `collStats` (page 423)
- `dbHash` (page 423)
- `dbStats` (page 423)
- `find` (page 419)
- `killCursors` (page 420)
- `listIndexes` (page 423)
- `listCollections` (page 423)

**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 291) 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 420)
- `dropIndex` (page 422)
- `emptycapped` (page 420)
- `find` (page 419)
- `insert` (page 419)
- `killCursors` (page 420)
- `listIndexes` (page 423)
- `listCollections` (page 423)
- `remove` (page 419)
- `renameCollectionSameDB` (page 423)
- `update` (page 419)

### Database Administration Roles

Every database includes the following database administration roles:

**dbAdmin**

Provides the following *actions* (page 419) on the database’s *system.indexes* (page 291), *system.namespaces* (page 291), and *system.profile* (page 291) collections:

- `collStats` (page 423)
- `dbHash` (page 423)
• *dbStats* (page 423)
• *find* (page 419)
• *killCursors* (page 420)
• *listIndexes* (page 423)
• *listCollections* (page 423)
• *dropCollection* (page 420) and *createCollection* (page 419) on *system.profile* (page 291) only

Changed in version 2.6.4: *dbAdmin* (page 406) added the *createCollection* (page 419) for the *system.profile* (page 291) collection. Previous versions only had the *dropCollection* (page 420) on the *system.profile* (page 291) 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 420)
• *dbStats* (page 423)
• *dropCollection* (page 420)
• *dropDatabase* (page 422)
• *dropIndex* (page 422)
• *enableProfiler* (page 420)
• *indexStats* (page 423)
• *reIndex* (page 422)
• *renameCollectionSameDB* (page 423)
• *repairDatabase* (page 423)
• *storageDetails* (page 421)
• *validate* (page 424)

**dbOwner**
The database owner can perform any administrative action on the database. This role combines the privileges granted by the *readWrite* (page 406), *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)
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 408), `clusterMonitor` (page 409), and `hostManager` (page 410) 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 422)
- `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 422)
- `splitChunk` (page 422)
- `splitVector` (page 422)

On the `config` database, provides the following actions on the `settings` (page 741) collection:

- `insert` (page 419)
- `remove` (page 419)
On the `config` database, provides the following actions on all configuration collections and on the `system.indexes` (page 291), `system.js` (page 291), and `system.namespaces` (page 291) 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 654) 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\footnote{https://cloud.mongodb.com/?jmp=docs} and Ops Manager\footnote{https://docs.opsmanager.mongodb.com/} monitoring agent.

Provides the following actions on the cluster as a whole:

- `connPoolStats` (page 423)
- `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 422)
- `netstat` (page 424)
- `replSetGetStatus` (page 421)
- `serverStatus` (page 424)
- `shardingState` (page 422)
- `top` (page 424)

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 291) collections in the cluster.

Provides the following actions on the `config` database’s configuration collections and `system.indexes` (page 291), `system.js` (page 291), and `system.namespaces` (page 291) 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 421)
  • `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\textsuperscript{83} backup agent, Ops Manager\textsuperscript{84} backup agent, or to use mongodump to back up an entire mongod instance.

Provides the following actions (page 419) 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 listCollections (page 423) action on all databases.

Provides the listIndexes (page 423) action for all collections.

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 291), system.namespaces (page 291), and system.js (page 291)
- the admin.system.users (page 291) and admin.system.roles (page 291) collections
- legacy system.users collections from versions of MongoDB prior to 2.6

To back up the system.profile (page 291) collection, which is created when you activate database profiling (page 215), you must have additional read access on this collection. Several roles provide this access, including the clusterAdmin (page 408) and dbAdmin (page 406) roles.

\textbf{restore}

Provides minimal privileges needed for restoring data from backups. This role provides sufficient privileges to use the mongorestore tool to restore an entire mongod instance.

Provides the following actions on all non-system collections and system.js (page 291) collections in the cluster; on the admin.system.users (page 291) and admin.system.roles (page 291) 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 420)
- dropCollection (page 420)
- insert (page 419)

Provides the listCollections (page 423) action on all databases.

Provides the following additional actions on admin.system.users (page 291) 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 291) collections in the cluster.

Although, restore (page 411) includes the ability to modify the documents in the admin.system.users (page 291) collection using normal modification operations, only modify these data using the user management methods.

\textsuperscript{83}https://cloud.mongodb.com/?jmp=docs
\textsuperscript{84}https://docs.opsmanager.mongodb.com/
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 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.

**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 421)
- listDatabases (page 423)

The role also provides the following actions on the admin.system.users (page 291) and admin.system.roles (page 291) 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)

**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 412) role

The following role provides full privileges on all resources:

root

Provides access to the operations and all the resources of the readWriteAnyDatabase (page 412), dbAdminAnyDatabase (page 412), userAdminAnyDatabase (page 412) and clusterAdmin (page 408) roles combined.

root (page 413) 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 291) collections in the admin database. root (page 413) 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 411) 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 applyOps commands, do not assign this role. Instead, create a user-defined role (page 388) that grants anyAction (page 424) on anyResource (page 419) and ensure that only the users who need access to these operations have this access.

system.roles Collection

New in version 2.6.

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:

```json
{
    _id: <system-defined id>,
    role: "<role name>",
    db: "<database>",
    privileges:
    [  
        { 
            resource: { <resource> },
            actions: [ "<action>", ... ]
        },
        ...
    ],
    roles:
...
```

6.4. Security Reference 413
A `system.roles` document has the following fields:

**admin.system.roles.role**
- The `role` field is a string that specifies the name of the role.

**admin.system.roles.db**
- The `db` 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`) and its database.

**admin.system.roles.privileges**
- The `privileges` array contains the privilege documents that define the privileges 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 apply. The document has one of the following form:

```
{ db: <database>, collection: <collection> }
```

or

```
{ cluster : true }
```

See **Resource Document** 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**.

**admin.system.roles.roles**
- The `roles` array contains role documents that specify the roles from which this role inherits 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** provided by MongoDB or a **user-defined role**.

**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:

```
```

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 418).
- 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 privilege permits actions on one system collections (page 291) 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:

```
```

6.4. Security Reference
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 418).

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.

The system.users collection in the admin database stores user authentication (page 320) and authorization (page 324) 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:

```json
{
    _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 372) 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 325).

A role document has the following syntax:

```json
{ 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 325).
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.

The customData (page 417) 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: {  
    "SCRAM-SHA-1": {  
      "iterationCount": 10000,  
      "salt": "nkHYXEZTTYmn+hrY994y1Q==",  
      "storedKey": "wxWGN3E1Q25WbPjACeXdUmN4nNo=";  
      "serverKey": "h7vBq5tACT/BtrIElY2QfTm+pQzM="  
    }  
  },  
  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 406) role in the test database, and the appUser role in the myApp database.

Resource Document

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 291). 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, system collections are excluded, unless you name them explicitly, as in the following:

```
{ db: "test", collection: "system.js" }
```

System collections include but are not limited to the following:

- `<database>.system.profile` (page 291)
- `<database>.system.js` (page 291)
- `system.users Collection` (page 416) in the admin database
- `system.roles Collection` (page 413) 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 291), 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.

Privilege actions define the operations a user can perform on a resource (page 417). A MongoDB privilege (page 325) 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 388).

**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 396).

`changeOwnPassword`

Users can change their own passwords. Apply this action to database resources. See also *Change Your Password and Custom Data* (page 396).

`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 \texttt{db.currentOp()} method to return pending and active operations. Apply this action to the \texttt{cluster} resource.

\textbf{invalidateUserCache}
Provides access to the \texttt{invalidateUserCache} command. Apply this action to the \texttt{cluster} resource.

\textbf{killOp}
User can perform the \texttt{db.killOp()} method. Apply this action to the \texttt{cluster} resource.

\textbf{planCacheRead}
User can perform the \texttt{planCacheListPlans} and \texttt{planCacheListQueryShapes} commands and the \texttt{PlanCache.getPlansByQuery()} and \texttt{PlanCache.listQueryShapes()} methods. Apply this action to database or collection resources.

\textbf{planCacheWrite}
User can perform the \texttt{planCacheClear} command and the \texttt{PlanCache.clear()} and \texttt{PlanCache.clearPlansByQuery()} methods. Apply this action to database or collection resources.

\textbf{storageDetails}
User can perform the \texttt{storageDetails} command. Apply this action to database or collection resources.

\textbf{Replication Actions}

\textbf{appendOplogNote}
User can append notes to the oplog. Apply this action to the \texttt{cluster} resource.

\textbf{replSetConfigure}
User can configure a replica set. Apply this action to the \texttt{cluster} resource.

\textbf{replSetGetStatus}
User can perform the \texttt{replSetGetStatus} command. Apply this action to the \texttt{cluster} resource.

\textbf{replSetHeartbeat}
User can perform the \texttt{replSetHeartbeat} command. Apply this action to the \texttt{cluster} resource.

\textbf{replSetStateChange}
User can change the state of a replica set through the \texttt{replSetFreeze}, \texttt{replSetMaintenance}, \texttt{replSetStepDown}, and \texttt{replSetSyncFrom} commands. Apply this action to the \texttt{cluster} resource.

\textbf{resync}
User can perform the \texttt{resync} command. Apply this action to the \texttt{cluster} resource.

\textbf{Sharding Actions}

\textbf{addShard}
User can perform the \texttt{addShard} command. Apply this action to the \texttt{cluster} resource.

\textbf{enableSharding}
User can enable sharding on a database using the \texttt{enableSharding} command and can shard a collection using the \texttt{shardCollection} command. Apply this action to database or collection resources.

\textbf{flushRouterConfig}
User can perform the \texttt{flushRouterConfig} command. Apply this action to the \texttt{cluster} resource.

\textbf{getShardMap}
User can perform the \texttt{getShardMap} command. Apply this action to the \texttt{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.

listCollections
User can perform the listCollections command. Apply this action to database resources.
**listIndexes**  
User can perform the `ListIndexes` command. Apply this action to database or collection resources.

**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 ports used by MongoDB:

<table>
<thead>
<tr>
<th>Default Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>27017</td>
<td>The default port for <code>mongod</code> and <code>mongos</code> instances. You can change this port with <code>port</code> or <code>--port</code>.</td>
</tr>
<tr>
<td>27018</td>
<td>The default port when running with <code>--shardsvr</code> runtime operation or the <code>shardsvr</code> value for the <code>clusterRole</code> setting in a configuration file.</td>
</tr>
<tr>
<td>27019</td>
<td>The default port when running with <code>--configsvr</code> runtime operation or the <code>configsvr</code> value for the <code>clusterRole</code> setting in a configuration file.</td>
</tr>
<tr>
<td>28017</td>
<td>The default port for the web status page. The web status page is always accessible at a port number that is 1000 greater than the port determined by <code>port</code>.</td>
</tr>
</tbody>
</table>

### System Event Audit Messages

**Note:** Available only in MongoDB Enterprise[^85].

**Audit Message**

The event auditing feature (page 330) can record events in JSON format. To configure auditing output, see **Configure System Events Auditing** (page 398).

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>
}

**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 324) 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.

<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
</table>
| authenticate | {  
|           |   user: <user name>,
|           |   db: <database>,
|           |   mechanism: <mechanism>  
|           | }                                | 0 - Success
|           |                                   | 18 - Authentication Failed |
| authCheck | {  
|           |   command: <name>,
|           |   ns: <database>.<collection>,
|           |   args: <command object>  
|           | }                                | 0 - Success
|           | ns field is optional.
|           | args field may be redacted.      | 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. 

---

86 Enabling auditAuthorizationSuccess degrades performance more than logging only the authorization failures.
<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>createCollection</td>
<td>{ ns: &lt;database&gt;.&lt;collection&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>createDatabase</td>
<td>{ ns: &lt;database&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>createIndex</td>
<td>{ ns: &lt;database&gt;.&lt;collection&gt;, indexName: &lt;index name&gt;, indexSpec: &lt;index specification&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>renameCollection</td>
<td>{ old: &lt;database&gt;.&lt;collection&gt;, new: &lt;database&gt;.&lt;collection&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>dropCollection</td>
<td>{ ns: &lt;database&gt;.&lt;collection&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>dropDatabase</td>
<td>{ ns: &lt;database&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>dropIndex</td>
<td>{ ns: &lt;database&gt;.&lt;collection&gt;, indexName: &lt;index name&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>createUser</td>
<td>{ user: &lt;user name&gt;, db: &lt;database&gt;, customData: &lt;document&gt;, roles: [ { role: &lt;role name&gt;, db: &lt;database&gt; }, ... ] }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>dropUser</td>
<td>{ user: &lt;user name&gt;, db: &lt;database&gt; }</td>
<td>0 - Success</td>
</tr>
</tbody>
</table>

The customData field is optional.
### Table 6.1 – continued from previous page

<table>
<thead>
<tr>
<th><strong>atype</strong></th>
<th><strong>param</strong></th>
<th><strong>result</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dropAllUsersFromDatabase</strong></td>
<td>{ db: <code>&lt;database&gt;</code> }</td>
<td>0 - Success</td>
</tr>
<tr>
<td><strong>updateUser</strong></td>
<td>{ user: <code>&lt;user name&gt;</code>, db: <code>&lt;database&gt;</code>, passwordChanged: <code>&lt;boolean&gt;</code>,</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td>customData: <code>&lt;document&gt;</code>, roles: [</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ role: <code>&lt;role name&gt;</code>, db: <code>&lt;database&gt;</code> }, ... ]</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>The customData field is optional.</strong></td>
<td></td>
</tr>
<tr>
<td><strong>grantRolesToUser</strong></td>
<td>{ user: <code>&lt;user name&gt;</code>, db: <code>&lt;database&gt;</code>, roles: [</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ role: <code>&lt;role name&gt;</code>, db: <code>&lt;database&gt;</code> }, ... ]</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td><strong>revokeRolesFromUser</strong></td>
<td>{ user: <code>&lt;user name&gt;</code>, db: <code>&lt;database&gt;</code>, roles: [</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ role: <code>&lt;role name&gt;</code>, db: <code>&lt;database&gt;</code> }, ... ]</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Table 6.1 – continued from previous page

<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>createRole</td>
<td>(page 420)</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>role: &lt;role name&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>db: &lt;database&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>roles: [</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>role: &lt;role name&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>db: &lt;database&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>},</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>],</td>
<td></td>
</tr>
<tr>
<td></td>
<td>privileges: [</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
<td></td>
</tr>
<tr>
<td></td>
<td>resource: &lt;resource</td>
<td></td>
</tr>
<tr>
<td></td>
<td>document&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>actions: [ &lt;action&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>... ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>},</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

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 419).

updateRole   | 0 - Success |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>role: &lt;role name&gt;,</td>
</tr>
<tr>
<td></td>
<td>db: &lt;database&gt;,</td>
</tr>
<tr>
<td></td>
<td>roles: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>role: &lt;role name&gt;,</td>
</tr>
<tr>
<td></td>
<td>db: &lt;database&gt;</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>],</td>
</tr>
<tr>
<td></td>
<td>privileges: [</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>resource: &lt;resource</td>
</tr>
<tr>
<td></td>
<td>document&gt;,</td>
</tr>
<tr>
<td></td>
<td>actions: [ &lt;action&gt;,</td>
</tr>
<tr>
<td></td>
<td>... ]</td>
</tr>
<tr>
<td></td>
<td>},</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

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 419).
Table 6.1 – continued from previous page

<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropRole (page 420)</td>
<td>{ role: &lt;role name&gt;, db: &lt;database&gt;</td>
<td>0 - Success</td>
</tr>
<tr>
<td>dropAllRolesFromDatabase</td>
<td>{ db: &lt;database&gt; }</td>
<td>0 - Success</td>
</tr>
<tr>
<td>grantRolesToRole</td>
<td>{ role: &lt;role name&gt;, db: &lt;database&gt;, roles: [ role: &lt;role name&gt;, db: &lt;database&gt; ], ... ]</td>
<td>0 - Success</td>
</tr>
<tr>
<td>revokeRolesFromRole</td>
<td>{ role: &lt;role name&gt;, db: &lt;database&gt;, roles: [ role: &lt;role name&gt;, db: &lt;database&gt; ], ... ]</td>
<td>0 - Success</td>
</tr>
<tr>
<td>grantPrivilegesToRole</td>
<td>{ role: &lt;role name&gt;, db: &lt;database&gt;, privileges: [ resource: &lt;resource document&gt;, actions: [ &lt;action&gt;, ... ] ], ... ]</td>
<td>0 - Success</td>
</tr>
</tbody>
</table>

For details on the resource document, see Resource Document (page 417). For a list of actions, see Privilege Actions (page 419).
<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>revokePrivilegesFromRole</td>
<td>{</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td>role: &lt;role name&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>db: &lt;database name&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>privileges: [</td>
<td></td>
</tr>
<tr>
<td></td>
<td>resource: &lt;resource document&gt;,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>actions: [ &lt;action&gt;, ... ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>},</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

For details on the resource document, see Resource Document (page 417). For a list of actions, see Privilege Actions (page 419).

| replSetReconfig               | {                                           | 0 - Success|
|                               |     old: <configuration>,                 |            |
|                               |     new: <configuration>                  |            |
|                               | }                                        |            |

Indicates membership change in the replica set. The old field is optional.

| enableSharding               | {                                           | 0 - Success|
|                               |     ns: <database> }                       |            |
|                               | }                                        |            |

| shardCollection              | {                                           | 0 - Success|
|                               |     ns: <database>.<collection>,           |            |
|                               |     key: <shard key pattern>,             |            |
|                               |     options: { unique: <boolean> }        |            |
|                               | }                                        |            |

| addShard                     | {                                           | 0 - Success|
|                               |     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.

| removeShard                  | {                                           | 0 - Success|
|                               |     shard: <shard name> }                  |            |
|                               | }                                        |            |

| shutdown                    | {                                           | 0 - Success|
|                               | }                                        |            |

Indicates commencement of database shutdown.
Table 6.1 – continued from previous page

<table>
<thead>
<tr>
<th>atype</th>
<th>param</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationMessage (page 422)</td>
<td>{ msg: '&lt;custom message string&gt;' }</td>
<td>0 - Success</td>
</tr>
<tr>
<td></td>
<td>See logApplicationMessage.</td>
<td></td>
</tr>
</tbody>
</table>

6.4.3 Security Release Notes Alerts


Security Release Notes

Access to system.users Collection

Changed in version 2.4.

In 2.4, only users with the userAdmin role have access to the system.users collection.

In version 2.2 and earlier, the read-write users of a database all have access to the system.users collection, which contains the user names and user password hashes.  

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

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 320), Enable Client Access Control (page 357), and Enable Authentication in a Sharded Cluster (page 358).

87 Read-only users do not have access to the system.users collection.
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 324), Create a Role (page 388), Create a User Administrator (page 384), and Add a User to a Database (page 386).

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 342).

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 335) and Configure Windows netsh Firewall for MongoDB (page 339).

6.5.5 Audit System Activity

Track access and changes to database configurations and data. MongoDB Enterprise™ 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 330) and Configure System Events Auditing (page 398).

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.

---

88http://www.mongodb.com/products/mongodb-enterprise?jmp=docs
6.5.8 Run MongoDB with Secure Configuration Options

MongoDB supports the execution of JavaScript code for certain server-side operations: mapReduce, group, 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[^89] 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[^90] to learn more about how you can use the key security capabilities to build compliant application infrastructure.

[^89]: http://www.mongodb.com/lp/contact/stig-requests
[^90]: http://info.mongodb.com/rs/mongodb/images/MongoDB_Security_Architecture_WP.pdf
Aggregation 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 440), 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 440) 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 452) Examples and tutorials for data aggregation operations in MongoDB.

**Aggregation Reference** (page 469) References for all aggregation operations material for all data aggregation methods in MongoDB.

### 7.1 Aggregation Introduction

**Aggregations** are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets. Running data aggregation on the `mongod` instance simplifies application code and limits resource requirements.

Like queries, aggregation operations in MongoDB use `collections` of documents as an input and return results in the form of one or more documents.
7.1.1 Aggregation Modalities

Aggregation Pipelines

MongoDB 2.2 introduced a new aggregation framework (page 440), 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. Aggregation Introduction 437
```
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"
}

distinct [ "A123", "B212" ]
```
7.1.2 Additional Features and Behaviors

Both the aggregation pipeline and map-reduce can operate on a sharded collection (page 661). Map-reduce operations can also output to a sharded collection. See Aggregation Pipeline and Sharded Collections (page 450) 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 474).

7.1.3 Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop
- The Aggregation Framework
- Webinar: Exploring the Aggregation Framework
- Quick Reference Cards

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 474) that compares the approaches.

**Aggregation Pipeline** (page 440) 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.

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.

```javascript
db.orders.aggregate([
  { $match: { status: "A" } },
  { $group: { _id: "$cust_id", total: [ $sum: "$amount" ] } }
])
```

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.

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 456) and Aggregation with the Zip Code Data Set (page 453).
Pipeline Expressions

Some pipeline stages take a pipeline expression as its operand. Pipeline expressions specify the transformation to apply to the input documents. Expressions have a document (page 168) structure and can contain other expression (page 470).

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 470).

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 450).
Map-Reduce

Map-reduce is a data processing paradigm for condensing large volumes of data into useful aggregated results. For map-reduce operations, MongoDB provides the mapReduce database command.

Consider the following map-reduce operation:

In this map-reduce operation, MongoDB applies the map phase to each input document (i.e. the documents in the collection that match the query condition). The map function emits key-value pairs. For those keys that have multiple values, MongoDB applies the reduce phase, which collects and condenses the aggregated data. MongoDB then stores the results in a collection. Optionally, the output of the reduce function may pass through a finalize function to further condense or process the results of the aggregation.

All map-reduce functions in MongoDB are JavaScript and run within the mongod process. Map-reduce operations take the documents of a single collection as the input and can perform any arbitrary sorting and limiting before beginning the map stage. mapReduce can return the results of a map-reduce operation as a document, or may write the results to collections. The input and the output collections may be sharded.

Note: For most aggregation operations, the Aggregation Pipeline (page 440) 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.
Collection →

```
db.orders.mapReduce(
  map → function() { emit( this.cust_id, this.amount ); },
  reduce → function(key, values) { return Array.sum( values ); },
  query → { status: "A" },
  output → { out: "order_totals" }
)
```

```
orders

{ 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"
}

query

map

reduce

```

{ cust_id: "A123",
  amount: 500,
  status: "A"
}

{ cust_id: "A123",
  amount: 250,
  status: "A"
}

{ cust_id: "B212",
  amount: 200,
  status: "A"
}

{ cust_id: "B212",
  amount: 300,
  status: "D"
}

```

reduce

```

{ "A123": [ 500, 250 ] }

{ "B212": [ 200 ] }

```

order_totals

```
that merge replace, merge, or reduce new results with previous results. See mapReduce and *Perform Incremental Map-Reduce* (page 463) for details and examples.

When returning the results of a map reduce operation *inline*, the result documents must be within the BSON Document Size limit, which is currently 16 megabytes. For additional information on limits and restrictions on map-reduce operations, see the [http://docs.mongodb.org/manual/reference/command/mapReduce](http://docs.mongodb.org/manual/reference/command/mapReduce) reference page.

MongoDB supports map-reduce operations on *sharded collections* (page 661). 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

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:

```json
{ 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:

```javascript
db.records.count()```

The following operation will count only the documents where the value of the field `a` is 1 and return 3:

```javascript
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:

```json
{ a: 1, b: 0 }
{ a: 1, b: 1 }
{ a: 1, b: 1 }
{ a: 1, b: 4 }
```

```javascript
```
```
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"
}

distinct [ "A123", "B212" ]
```
Consider the following `db.collection.distinct()` operation which returns the distinct values of the field `b`:

```javascript
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:

```javascript
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 440).
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 450) Mechanics of aggregation pipeline operations on sharded collections.


*Map Reduce Concurrency* (page 452) Details the locks taken during map-reduce operations.

### Aggregation Pipeline Optimization

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:

```json
{ $sort: { age : -1 } },
{ $match: { status: 'A' } }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```json
{ $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:

```json
{ $skip: 10 },
{ $limit: 5 }
```

During the optimization phase, the optimizer transforms the sequence to the following:

```json
{ $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 450), 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:

```json
{ $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:

```json
{ $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. See sort-and-memory for more information.

$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:

```json
{ $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.

```json
{ $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:

```json
{ $skip: 5 },
{ $skip: 2 }
```

$sort + $limit + $skip + $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. See sort-and-memory for more information.

$limit + $limit + $skip + $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:

```json
{ $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.

```json
{ $limit: 10 }
```

$skip + $skip + $limit 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:

```json
{ $skip: 5 },
{ $skip: 2 }
```

$sort + $skip + $sort Coalescence When a $sort immediately follows another $sort, the two stages can coalesce into a single $sort where the sort amount is the sum of the two initial sort amounts. For example, a pipeline contains the following sequence:

```json
{ $sort: [ "field1", "field2" ] },
{ $sort: [ "field2", "field1" ] }
```

Then the second $sort stage can coalesce into the first $sort stage and result in a single $sort stage where the sort amount is the sum of the two initial sort amounts.

```json
{ $sort: [ "field1", "field2" ] }
```

$skip + $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:

```json
{ $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 minimum of the two initial skip amounts 5 and 2.

```json
{ $skip: 7 }
```

$sort + $sort + $sort Coalescence When a $sort immediately follows another $sort, the two stages can coalesce into a single $sort where the sort amount is the sum of the two initial sort amounts. For example, a pipeline contains the following sequence:

```json
{ $sort: [ "field1", "field2" ] },
{ $sort: [ "field2", "field1" ] }
```

Then the second $sort stage can coalesce into the first $sort stage and result in a single $sort stage where the sort amount is the sum of the two initial sort amounts.

```json
{ $sort: [ "field1", "field2" ] }
```
Then the second `$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 447) to transforms the sequence to the following:

```
{ $sort: { age : -1 } },
{ $limit: 15 }
{ $skip: 10 }
```

The `$skip + $limit Sequence Optimization` (page 447) increases the `$limit` amount with the reordering. See `$skip + $limit Sequence Optimization` (page 447) 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 + $limit Sequence Optimization` (page 447) reverses the position of the { `$skip` 5 } and { `$limit` 10 } stages and increases the limit amount.
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:

```json
{ $limit: 15 },
{ $skip: 7 }
```

See $limit + $limit Coalescence (page 448) and $skip + $skip Coalescence (page 448) for details.

See also:
```
explain option in the db.collection.aggregate()
```

## Aggregation Pipeline Limits

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

The aggregation pipeline supports operations on `sharded` collections. This section describes behaviors specific to the aggregation pipeline (page 440) 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 669). 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.\(^{12}\)

**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**

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.

\(^{12}\) Until all shards upgrade to v2.6, the second pipeline runs on the mongos if any shards are still running v2.4.

#### 7.2. Aggregation Concepts
• `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 456](#)) Use the pipeline to sort, normalize, and sum data on a collection of user data.

**Map-Reduce Examples** ([page 460](#)) Define map-reduce operations that select ranges, group data, and calculate sums and averages.

**Perform Incremental Map-Reduce** ([page 463](#)) Run a map-reduce operations over one collection and output results to another collection.

**Troubleshoot the Map Function** ([page 465](#)) Steps to troubleshoot the `map` function.

**Troubleshoot the Reduce Function** ([page 466](#)) Steps to troubleshoot the `reduce` function.
7.3.1 Aggregation with the Zip Code Data Set

The examples in this document use the zipcodes collection. This collection is available at: media.mongodb.org/zips.json\(^1\). Use mongoimport to load this data set into your mongod instance.

Data Model

Each document in the zipcodes collection has the following form:

```
{
   "_id": "10280",
   "city": "NEW YORK",
   "state": "NY",
   "pop": 5574,
   "loc": [
      -74.016323,
      40.710537
   ]
}
```

- The _id field holds the zip code as a string.
- The city field holds the city 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 440) to processes documents into aggregated results. An aggregation pipeline (page 440) 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 440) 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.

\(^1\)http://media.mongodb.org/zips.json
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 440) 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.  

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:

---

\[^{14}\] 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.
See also:
$group,$sum,$avg

Return Largest and Smallest Cities by State

The following aggregation operation returns the smallest and largest cities by population for each state:

```javascript
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 440) 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": "EDGEBASE"
   },
   "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" : "BENGEE",
      "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

#### 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.

```javascript
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:

```json
[
  { "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.

```javascript
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`. 

7.3. Aggregation Examples
- 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.

```javascript
db.users.aggregate(
    [
        { $project : { month_joined : { $month : "joined" } } },
        { $group : { _id : "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:
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.

```javascript
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:

```javascript
{  
  _id : "jane",  
  joined : ISODate("2011-03-02"),  
  likes : ["golf", "racquetball"]  
}
```

The `$unwind` operator would create the following documents:

```javascript
{  
  _id : "jane",  
  joined : ISODate("2011-03-02"),  
  likes : "golf"  
}
```

```javascript
{  
  _id : "jane",  
  joined : ISODate("2011-03-02"),  
  likes : "racquetball"  
}
```
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:

```json
{
  "_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

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:

```json
{
  _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 \texttt{orders} collection to group by the \texttt{cust_id}, and calculate the sum of the price for each \texttt{cust_id}:

1. Define the map function to process each input document:
   - In the function, \texttt{this} refers to the document that the map-reduce operation is processing.
   - The function maps the price to the \texttt{cust_id} for each document and emits the \texttt{cust_id} and price pair.

   \begin{verbatim}
   var mapFunction1 = function() {
       emit(this.cust_id, this.price);
   };
   \end{verbatim}

2. Define the corresponding reduce function with two arguments \texttt{keyCustId} and \texttt{valuesPrices}:
   - The \texttt{valuesPrices} is an array whose elements are the price values emitted by the map function and grouped by \texttt{keyCustId}.
   - The function reduces the \texttt{valuesPrice} array to the sum of its elements.

   \begin{verbatim}
   var reduceFunction1 = function(keyCustId, valuesPrices) {
       return Array.sum(valuesPrices);
   };
   \end{verbatim}

3. Perform the map-reduce on all documents in the \texttt{orders} collection using the \texttt{mapFunction1} map function and the \texttt{reduceFunction1} reduce function.

   \begin{verbatim}
   db.orders.mapReduce(
       mapFunction1,
       reduceFunction1,
       { out: "map_reduce_example" }
   )
   \end{verbatim}

   This operation outputs the results to a collection named \texttt{map_reduce_example}. If the \texttt{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 \texttt{orders} collection for all documents that have an \texttt{ord_date} value greater than 01/01/2012. The operation groups by the \texttt{item.sku} field, and calculates the number of orders and the total quantity ordered for each \texttt{sku}. The operation concludes by calculating the average quantity per order for each \texttt{sku} value:

1. Define the map function to process each input document:
   - In the function, \texttt{this} refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the \texttt{sku} with a new object \texttt{value} that contains the \texttt{count} of 1 and the item \texttt{qty} for the order and emits the \texttt{sku} and \texttt{value} pair.

   \begin{verbatim}
   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);
       }
   };
   \end{verbatim}
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.

```javascript
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:

```javascript
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.

```javascript
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

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:

```javascript
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`:
   ```javascript
   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`.
```javascript
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.

```javascript
var finalizeFunction = function(key, reducedValue) {
  if (reducedValue.count > 0)
    reducedValue.avg_time = reducedValue.total_time / reducedValue.count;

  return reducedValue;
};
```

4. Perform map-reduce on the sessions collection using the mapFunction, the reduceFunction, and the finalizeFunction functions. Output the results to a collection session_stat. If the session_stat collection already exists, the operation will replace the contents:

```javascript
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:

```javascript
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:

```javascript
db.sessions.mapReduce( mapFunction,
reduceFunction,
{
```
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:

```javascript
{
    _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:

```javascript
var map = function() {
    emit(this.cust_id, this.price);
};
```

2. Define the emit function to print the key and value:

```javascript
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:

```javascript
var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
map.apply(myDoc);
```

4. Verify the key and value pair is as you expected.

```javascript
emit
key: abc123 value:250
```

5. Invoke the map function with multiple documents from the orders collection:

```javascript
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

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:

   ```javascript
   var reduceFunction1 = function(keyCustId, valuesPrices) {
     return Array.sum(valuesPrices);
   };
   ```

2. Define a sample array of integers:

   ```javascript
   var myTestValues = [ 5, 5, 10 ];
   ```

3. Invoke the `reduceFunction1` with `myTestValues`:

   ```javascript
   reduceFunction1('myKey', myTestValues);
   ```

4. Verify the `reduceFunction1` returned an integer:

   ```javascript
   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`:

   ```javascript
   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:

```javascript
var myTestObjects = [
    { count: 1, qty: 5 },
    { count: 2, qty: 10 },
    { count: 3, qty: 15 }
];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```javascript
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```javascript
{ "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:

```javascript
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`:

```javascript
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`:

```javascript
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```javascript
{ "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:

   ```javascript
   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:

   ```javascript
   var myKey = 'myKey';
   ```

3. Define a sample valuesIdempotent array that contains an element that is a call to the reduceFunction2 function:

   ```javascript
   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:

   ```javascript
   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:

   ```javascript
   reduceFunction2(myKey, valuesIdempotent);
   reduceFunction2(myKey, values1);
   ```

6. Verify the reduceFunction2 returned the same result:

   ```javascript
   { "count" : 6, "qty" : 30 }
   ```

7.3.7 Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data

• MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop\textsuperscript{16}
• The Aggregation Framework\textsuperscript{17}
• Webinar: Exploring the Aggregation Framework\textsuperscript{18}
• Quick Reference Cards\textsuperscript{19}

7.4 Aggregation Reference

Aggregation Pipeline Quick Reference \textsuperscript{(page 469)} 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 \textsuperscript{(page 474)} A comparison of group, mapReduce and aggregate that explores the strengths and limitations of each aggregation modality.

SQL to Aggregation Mapping Chart \textsuperscript{(page 476)} An overview common aggregation operations in SQL and MongoDB using the aggregation pipeline and operators in MongoDB and common SQL statements.

Aggregation Commands \textsuperscript{(page 478)} The reference for the data aggregation commands, which provide the interfaces to MongoDB’s aggregation capability.

Variables in Aggregation Expressions \textsuperscript{(page 478)} Use of variables in aggregation pipeline expressions.

7.4.1 Aggregation Pipeline Quick Reference

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.

\begin{verbatim}
  db.collection.aggregate( [ { <stage> }, ... ] )
\end{verbatim}

\textsuperscript{17}https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs
\textsuperscript{18}https://www.mongodb.com/webinar/exploring-the-aggregation-framework?jmp=docs
\textsuperscript{19}https://www.mongodb.com/lp/misc/quick-reference-cards?jmp=docs
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$project</td>
<td>Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.</td>
</tr>
<tr>
<td>$match</td>
<td>Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).</td>
</tr>
<tr>
<td>$redact</td>
<td>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 $match. Can be used to implement field level redaction. For each input document, outputs either one or zero document.</td>
</tr>
<tr>
<td>$limit</td>
<td>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).</td>
</tr>
<tr>
<td>$skip</td>
<td>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).</td>
</tr>
<tr>
<td>$unwind</td>
<td>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.</td>
</tr>
<tr>
<td>$group</td>
<td>Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.</td>
</tr>
<tr>
<td>$sort</td>
<td>Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.</td>
</tr>
<tr>
<td>$geoNear</td>
<td>Returns an ordered stream of documents based on the proximity to a geospatial point. Incorporates the functionality of $match, $sort, and $limit for geospatial data. The output documents include an additional distance field and can include a location identifier field.</td>
</tr>
<tr>
<td>$out</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Expressions**

Expressions can include field paths and system variables (page 470), literals (page 471), expression objects (page 471), and expression operators (page 471). 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 479) 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 479) can be rebound.

Along with the CURRENT (page 479) system variable, other system variables (page 478) 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 471) 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$and</td>
<td>Returns true only when <em>all</em> its expressions evaluate to true. Accepts any number of argument expressions.</td>
</tr>
<tr>
<td>$or</td>
<td>Returns true when <em>any</em> of its expressions evaluates to true. Accepts any number of argument expressions.</td>
</tr>
<tr>
<td>$not</td>
<td>Returns the boolean value that is the opposite of its argument expression. Accepts a single argument expression.</td>
</tr>
</tbody>
</table>

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.
### 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 179) for values of different types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$cmp</code></td>
<td>Returns: 0 if the two values are equivalent, 1 if the first value is greater than the second, and −1 if the first value is less than the second.</td>
</tr>
<tr>
<td><code>$eq</code></td>
<td>Returns <code>true</code> if the values are equivalent.</td>
</tr>
<tr>
<td><code>$gt</code></td>
<td>Returns <code>true</code> if the first value is greater than the second.</td>
</tr>
<tr>
<td><code>$gte</code></td>
<td>Returns <code>true</code> if the first value is greater than or equal to the second.</td>
</tr>
<tr>
<td><code>$lt</code></td>
<td>Returns <code>true</code> if the first value is less than the second.</td>
</tr>
<tr>
<td><code>$lte</code></td>
<td>Returns <code>true</code> if the first value is less than or equal to the second.</td>
</tr>
<tr>
<td><code>$ne</code></td>
<td>Returns <code>true</code> if the values are not equivalent.</td>
</tr>
</tbody>
</table>

### Arithmetic Expressions
Arithmetic expressions perform mathematic operations on numbers. Some arithmetic expressions can also support date arithmetic.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$add</code></td>
<td>Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.</td>
</tr>
<tr>
<td><code>$subtract</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>$multiply</code></td>
<td>Multiplies numbers to return the product. Accepts any number of argument expressions.</td>
</tr>
<tr>
<td><code>$divide</code></td>
<td>Returns the result of dividing the first number by the second. Accepts two argument expressions.</td>
</tr>
<tr>
<td><code>$mod</code></td>
<td>Returns the remainder of the first number divided by the second. Accepts two argument expressions.</td>
</tr>
</tbody>
</table>

### 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.

---

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$concat</code></td>
<td>Concatenates any number of strings.</td>
</tr>
<tr>
<td><code>$substr</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>$toLowerCase</code></td>
<td>Converts a string to lowercase. Accepts a single argument expression.</td>
</tr>
<tr>
<td><code>$toUpperCase</code></td>
<td>Converts a string to uppercase. Accepts a single argument expression.</td>
</tr>
<tr>
<td><code>$strCase</code></td>
<td>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.</td>
</tr>
</tbody>
</table>

**Text Search Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$meta</code></td>
<td>Access text search metadata.</td>
</tr>
</tbody>
</table>

**Array Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$size</code></td>
<td>Returns the number of elements in the array. Accepts a single expression as argument.</td>
</tr>
</tbody>
</table>

**Variable Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$map</code></td>
<td>Applies a subexpression to each element of an array and returns the array of resulting values in order. Accepts named parameters.</td>
</tr>
<tr>
<td><code>$let</code></td>
<td>Defines variables for use within the scope of a subexpression and returns the result of the subexpression. Accepts named parameters.</td>
</tr>
</tbody>
</table>

**Literal Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$literal</code></td>
<td>Return a value without parsing. Use for values that the aggregation pipeline may interpret as an expression. For example, use a <code>$literal</code> expression to a string that starts with a <code>$</code> to avoid parsing as a field path.</td>
</tr>
</tbody>
</table>

**Date Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$dayOfYear</code></td>
<td>Returns the day of the year for a date as a number between 1 and 366 (leap year).</td>
</tr>
<tr>
<td><code>$dayOfMonth</code></td>
<td>Returns the day of the month for a date as a number between 1 and 31.</td>
</tr>
<tr>
<td><code>$dayOfWeek</code></td>
<td>Returns the day of the week for a date as a number between 1 (Sunday) and 7 (Saturday).</td>
</tr>
<tr>
<td><code>$year</code></td>
<td>Returns the year for a date as a number (e.g. 2014).</td>
</tr>
<tr>
<td><code>$month</code></td>
<td>Returns the month for a date as a number between 1 (January) and 12 (December).</td>
</tr>
<tr>
<td><code>$week</code></td>
<td>Returns the week number for a date as a number between 0 (the partial week that precedes the first Sunday of the year) and 53 (leap year).</td>
</tr>
<tr>
<td><code>$hour</code></td>
<td>Returns the hour for a date as a number between 0 and 23.</td>
</tr>
<tr>
<td><code>$minute</code></td>
<td>Returns the minute for a date as a number between 0 and 59.</td>
</tr>
<tr>
<td><code>$second</code></td>
<td>Returns the seconds for a date as a number between 0 and 60 (leap seconds).</td>
</tr>
<tr>
<td><code>$milliseconds</code></td>
<td>Returns the milliseconds of a date as a number between 0 and 999.</td>
</tr>
<tr>
<td><code>$dateToString</code></td>
<td>Returns the date as a formatted string.</td>
</tr>
</tbody>
</table>

**Conditional Expressions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$cond</code></td>
<td>A ternary operator that evaluates one expression, and depending on the result, returns the value of one of the other two expressions. Accepts either three expressions in an ordered list or three named parameters.</td>
</tr>
<tr>
<td><code>$ifNull</code></td>
<td>Returns either the non-null result of the first expression or the result of the second expression if the first expression results in a null result. Null result encompasses instances of undefined values or missing fields. Accepts two expressions as arguments. The result of the second expression can be null.</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sum</td>
<td>Returns a sum for each group. Ignores non-numeric values.</td>
</tr>
<tr>
<td>$avg</td>
<td>Returns an average for each group. Ignores non-numeric values.</td>
</tr>
<tr>
<td>$first</td>
<td>Returns a value from the first document for each group. Order is only defined if the documents are in a defined order.</td>
</tr>
<tr>
<td>$last</td>
<td>Returns a value from the last document for each group. Order is only defined if the documents are in a defined order.</td>
</tr>
<tr>
<td>$max</td>
<td>Returns the highest expression value for each group.</td>
</tr>
<tr>
<td>$min</td>
<td>Returns the lowest expression value for each group.</td>
</tr>
<tr>
<td>$push</td>
<td>Returns an array of expression values for each group.</td>
</tr>
<tr>
<td>$addToSet</td>
<td>Returns an array of unique expression values for each group. Order of the array elements is undefined.</td>
</tr>
</tbody>
</table>

7.4.2 Aggregation Commands Comparison

The following table provides a brief overview of the features of the MongoDB aggregation commands.
<table>
<thead>
<tr>
<th>Description</th>
<th>aggregate</th>
<th>mapReduce</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 $sort. See <a href="http://docs.mongodb.org/manual/reference/operator/aggregation">http://docs.mongodb.org/manual/reference/operator/aggregation</a> for more information on the pipeline operators.</td>
<td>Implements the Map-Reduce aggregation for processing large data sets.</td>
<td>Provides grouping functionality. Is slower than the <code>aggregate</code> command and has less functionality than the <code>mapReduce</code> command.</td>
<td></td>
</tr>
<tr>
<td>Key Features</td>
<td>Pipeline operators can be repeated as needed. Pipeline operators need not produce one output document for every input document. Can also generate new documents or filter out documents.</td>
<td>In addition to grouping operations, can perform complex aggregation tasks as well as perform incremental aggregation on continuously growing datasets. See Map-Reduce Examples (page 460) and Perform Incremental Map-Reduce (page 463). Custom <code>map</code>, <code>reduce</code> and finalize JavaScript functions offer flexibility to aggregation logic. See <code>mapReduce</code> for details and restrictions on the functions.</td>
<td>Can either group by existing fields or with a custom <code>keyf</code> JavaScript function, can group by calculated fields. See <code>group</code> for information and example using the <code>keyf</code> function.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Limited to the operators and expressions supported by the aggregation pipeline. However, can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results by using the <code>$project</code> pipeline operator. See <code>$project</code> for more information as well as <a href="http://docs.mongodb.org/manual/reference/operator/aggregation">http://docs.mongodb.org/manual/reference/operator/aggregation</a> for more information on all the available pipeline operators.</td>
<td>Returns results in various options (inline, new collection, merge, replace, reduce). See <code>mapReduce</code> for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.</td>
<td>Custom <code>reduce</code> and finalize JavaScript functions offer flexibility to grouping logic. See <code>group</code> for details and restrictions on these functions.</td>
</tr>
<tr>
<td>Output Results</td>
<td>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.</td>
<td>Returns results in various options (inline, new collection, merge, replace, reduce). See <code>mapReduce</code> for details on the output options. Changed in version 2.2: Provides much better support for sharded map-reduce output than previous versions.</td>
<td>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.</td>
</tr>
<tr>
<td>Sharding Notes</td>
<td>Supports non-sharded and sharded input collections.</td>
<td>Supports non-sharded and sharded input collections. Prior to 2.4, JavaScript code executed in a single thread. See Map-Reduce (page 442) and <code>mapReduce</code>.</td>
<td>Does not support sharded collection. Prior to 2.4, JavaScript code executed in a single thread. See <code>group</code>.</td>
</tr>
<tr>
<td>More Information</td>
<td>See Aggregation Pipeline (page 440) and <code>aggregate</code>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.4.3 SQL to Aggregation Mapping Chart

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:

<table>
<thead>
<tr>
<th>SQL Terms, Functions, and Concepts</th>
<th>MongoDB Aggregation Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$match</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>$group</td>
</tr>
<tr>
<td>HAVING</td>
<td>$match</td>
</tr>
<tr>
<td>SELECT</td>
<td>$project</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>$sort</td>
</tr>
<tr>
<td>LIMIT</td>
<td>$limit</td>
</tr>
<tr>
<td>SUM()</td>
<td>$sum</td>
</tr>
<tr>
<td>COUNT()</td>
<td>$sum</td>
</tr>
<tr>
<td>join</td>
<td>No direct corresponding operator; however, the $unwind operator allows for somewhat similar functionality, but with fields embedded within the document.</td>
</tr>
</tbody>
</table>

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:

```json
{
    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 }]
}
```
<table>
<thead>
<tr>
<th>SQL Example</th>
<th>MongoDB Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT COUNT(*) AS count FROM orders</strong></td>
<td><code>db.orders.aggregate( [ </code></td>
<td>Count all records from orders</td>
</tr>
<tr>
<td><code>} </code></td>
<td><code>{ $group: {</code>_id: null, count: { $sum: 1 } } } ] )`</td>
<td></td>
</tr>
<tr>
<td><strong>SELECT SUM(price) AS total FROM orders</strong></td>
<td><code>db.orders.aggregate( [ </code></td>
<td>Sum the price field from orders</td>
</tr>
<tr>
<td><code>} </code></td>
<td><code>{ $group: {</code>_id: null, total: { $sum: &quot;$price&quot; } } } ] )`</td>
<td></td>
</tr>
<tr>
<td><strong>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id</strong></td>
<td><code>db.orders.aggregate( [ </code></td>
<td>For each unique cust_id, sum the price field.</td>
</tr>
<tr>
<td><code>} </code></td>
<td><code>{ $group: {</code>_id: &quot;$cust_id&quot;, total: { $sum: &quot;$price&quot; } } } ] )`</td>
<td></td>
</tr>
<tr>
<td><strong>SELECT cust_id, SUM(price) AS total FROM orders GROUP BY cust_id ORDER BY total</strong></td>
<td><code>db.orders.aggregate( [ </code></td>
<td>For each unique cust_id, sum the price field, results sorted by sum.</td>
</tr>
<tr>
<td><code>} </code></td>
<td><code>{ $group: {</code>_id: &quot;$cust_id&quot;, total: { $sum: &quot;$price&quot; } } }, `</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code> { $sort: { total: 1 } } ] )</code></td>
<td></td>
</tr>
<tr>
<td><strong>SELECT cust_id, ord_date, SUM(price) AS total FROM orders GROUP BY cust_id, ord_date</strong></td>
<td><code>db.orders.aggregate( [ </code></td>
<td>For each unique cust_id, ord_date grouping, sum the price field. Excludes the time portion of the date.</td>
</tr>
<tr>
<td><code>} </code></td>
<td><code>{ $group: {</code>_id: { cust_id: &quot;$cust_id&quot;, ord_date: { month: { $month: &quot;$ord_date&quot; }, day: { $dayOfMonth: &quot;$ord_date&quot; }, year: { $year: &quot;$ord_date&quot; } } }, total: { $sum: &quot;$price&quot; } } } ] )`</td>
<td></td>
</tr>
</tbody>
</table>

### 7.4. Aggregation Reference

**SELECT cust_id, count(*) FROM orders** | `db.orders.aggregate( [ ` | For cust_id with multiple records, return the cust_id and the corresponding record count. |
| `} ` | ` { $group: { } } ] )` |
7.4.4 Aggregation Commands

Aggregation Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aggregate</td>
<td>Performs aggregation tasks (page 439) such as group using the aggregation framework.</td>
</tr>
<tr>
<td>count</td>
<td>Counts the number of documents in a collection.</td>
</tr>
<tr>
<td>distinct</td>
<td>Displays the distinct values found for a specified key in a collection.</td>
</tr>
<tr>
<td>group</td>
<td>Groups documents in a collection by the specified key and performs simple aggregation.</td>
</tr>
<tr>
<td>mapReduce</td>
<td>Performs map-reduce (page 442) aggregation for large data sets.</td>
</tr>
</tbody>
</table>

Aggregation Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.collection.aggregate()</td>
<td>Provides access to the aggregation pipeline (page 440).</td>
</tr>
<tr>
<td>db.collection.group()</td>
<td>Groups documents in a collection by the specified key and performs simple aggregation.</td>
</tr>
<tr>
<td>db.collection.mapReduce()</td>
<td>Performs map-reduce (page 442) aggregation for large data sets.</td>
</tr>
</tbody>
</table>

7.4.5 Variables in Aggregation Expressions

Aggregation expressions (page 470) can use both user-defined and system variables.

Variables can hold any BSON type data (page 178). 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>"."

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:

---

22http://www.mongodb.com/mongodb-and-mysql-compared?jmp=docs
24https://www.mongodb.com/products/consulting?jmp=docs#database_modernization
### Variable

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROOT</td>
<td>References the root document, i.e. the top-level document, currently being processed in the aggregation pipeline stage. References the start of the field path being processed in the aggregation pipeline stage. Unless documented otherwise, all stages start with CURRENT (page 479) the same as ROOT (page 479). CURRENT (page 479) is modifiable. However, since $&lt;field&gt;$ is equivalent to $$CURRENT.&lt;field&gt;$$, rebinding CURRENT (page 479) changes the meaning of $ accesses. One of the allowed results of a $redact$ expression.</td>
</tr>
<tr>
<td>CURRENT</td>
<td>One of the allowed results of a $redact$ expression.</td>
</tr>
<tr>
<td>DESCEND</td>
<td>One of the allowed results of a $redact$ expression.</td>
</tr>
<tr>
<td>PRUNE</td>
<td>One of the allowed results of a $redact$ expression.</td>
</tr>
<tr>
<td>KEEP</td>
<td>One of the allowed results of a $redact$ expression.</td>
</tr>
</tbody>
</table>

See also:

$let, $redact, $map

### 7.5 Additional Resources

- MongoDB Analytics: Learn Aggregation by Example: Exploratory Analytics and Visualization Using Flight Data\(^5\)
- MongoDB for Time Series Data: Analyzing Time Series Data Using the Aggregation Framework and Hadoop\(^6\)
- The Aggregation Framework\(^7\)
- Webinar: Exploring the Aggregation Framework\(^8\)
- Quick Reference Cards\(^9\)

---


\(^7\) [https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs](https://www.mongodb.com/presentations/aggregation-framework-0?jmp=docs)


Indexes provide high performance read operations for frequently used queries.

This section introduces indexes in MongoDB, describes the types and configuration options for indexes, and describes special types of indexing MongoDB supports. The section also provides tutorials detailing procedures and operational concerns, and providing information on how applications may use indexes.

**Index Introduction** (page 481) An introduction to indexes in MongoDB.

**Index Concepts** (page 486) The core documentation of indexes in MongoDB, including geospatial and text indexes.

**Index Types** (page 486) MongoDB provides different types of indexes for different purposes and different types of content.

**Index Properties** (page 503) The properties you can specify when building indexes.

**Index Creation** (page 507) The options available when creating indexes.

**Index Intersection** (page 510) The use of index intersection to fulfill a query.

**Indexing Tutorials** (page 517) Examples of operations involving indexes, including index creation and querying indexes.

**Indexing Reference** (page 552) Reference material for indexes in MongoDB.

### 8.1 Index Introduction

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.

---

\(^1\) MongoDB indexes use a B-tree data structure.
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.

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 548) 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 548) 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 511) for more information on multikey indexes.

**Geospatial Index**

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: 2d indexes (page 497) that uses planar geometry when returning results and 2sphere indexes (page 496) that use spherical geometry to return results.

See 2d Index Internals (page 499) 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 500) for more information on text indexes and search.

Hashed Indexes

To support hash based sharding (page 674), MongoDB provides a hashed index (page 502) 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 504) property for an index causes MongoDB to reject duplicate values for the indexed field. Other than the unique constraint, unique indexes are functionally interchangeable with other MongoDB indexes.

Sparse Indexes

The sparse (page 505) 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 503) 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 211) for implementation instructions.

### 8.1.3 Index Use

Indexes can improve the efficiency of read operations. The *Analyze Query Performance* (page 111) 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 68).

### 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.

```javascript
db.users.find( { score: { "$lt": 30 } }, { score: 1, _id: 0 } )
```

For more information on covered queries, see *Covered Query* (page 66).

### 8.1.5 Index Intersection

New in version 2.6.

MongoDB can use the *intersection of indexes* (page 510) 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 510).

### 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 517). The Indexing Reference (page 552) documents the commands and operations specific to index construction, maintenance, and querying in MongoDB, including index types and creation options.

Index Types (page 486) MongoDB provides different types of indexes for different purposes and different types of content.

- 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 500) Text indexes support search of string content in documents.

- Hashed Index (page 502) 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.

Index Properties (page 503) The properties you can specify when building indexes.

- TTL Indexes (page 503) The TTL index is used for TTL collections, which expire data after a period of time.

- Unique Indexes (page 504) A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

- Sparse Indexes (page 505) A sparse index does not index documents that do not have the indexed field.

Index Creation (page 507) The options available when creating indexes.

Index Intersection (page 510) The use of index intersection to fulfill a query.

Multikey Index Bounds (page 511) The computation of bounds on a multikey index scan.

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 createIndex() method. For more detailed instructions about building indexes, see the Indexing Tutorials (page 517) 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 500) Text indexes support search of string content in documents.

**Hashed Index** (page 502) 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**

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.

![Index Illustration](image)

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:

```json
{   
    "_id" : ObjectId(...), 
    "name" : "Alice", 
    "age" : 27 
}
```

The following command creates an index on the name field:

```javascript
db.friends.createIndex( { "name" : 1 } )
```

**Cases**

**_id Field Index**  MongoDB creates the _id index, which is an ascending unique index (page 504) 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 209) 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:

```json
{_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:

```javascript
db.people.createIndex( { "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:

```json
{_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:

```javascript
db.factories.createIndex( { metro: 1 } )
```

The following query can use the index on the metro field:

```javascript
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**

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 index diagram](image)

\(\{ \text{userid: 1, score: -1} \}\) Index

Compound indexes can support queries that match on multiple fields.

**Example**

Consider a collection named products that holds documents that resemble the following document:

```json
{ 
  "_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.createIndex( { "item": 1, "stock": 1 } )
```

**Important:** You may not create compound indexes that have hashed index fields. You will receive an error if you attempt to create a compound index that includes a hashed index (page 502).

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.

\(^2\)MongoDB imposes a limit of 31 fields for any compound index.
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:

```javascript
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:

```javascript
db.events.find().sort( { username: -1, date: 1 } )
```

The following index can support both these sort operations:

```javascript
db.events.createIndex( { "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:

```javascript
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 548).

**Prefixes**

Index prefixes are the *beginning* subsets of indexed fields. For example, consider the following compound index:

```javascript
{ "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 510) 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 510) for more details.

Multikey Indexes

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:

```javascript
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 511).

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:

```javascript
{ _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:

```javascript
{ _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 502) indexes *cannot* be multikey.

Examples

**Index Basic Arrays** Consider a *survey* collection with the following document:

```javascript
{ _id: 1, item: "ABC", ratings: [ 2, 5, 9 ] }
```

Create an index on the field `ratings`:

```javascript
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 511). 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 548).
Geospatial Indexes and Queries

MongoDB offers a number of indexes and query mechanisms to handle geospatial information. This section introduces MongoDB’s geospatial features. For complete examples of geospatial queries in MongoDB, see Geospatial Index Tutorials (page 529).

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 496) 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 497) 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 496) 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 531)

**2d**  2d (page 497) 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 534)

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 496)  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 497)  The 2d index supports data stored as legacy coordinate pairs and is intended for use in MongoDB 2.2 and earlier.

- **geoHaystack Indexes** (page 498)  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 499)  Provides a more in-depth explanation of the internals of geospatial indexes. This material is not necessary for normal operations but may be useful for troubleshooting and for further understanding.

**2dsphere Indexes**  New in version 2.4.

**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](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 553) objects and as legacy coordinate pairs (See also [2dsphere Indexed Field Restrictions](page 497)). For legacy coordinate pairs, the index converts the data to GeoJSON Point (page 553). For details on the supported GeoJSON objects, see [GeoJSON Objects](page 553).

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 505) by default and ignores the `sparse: true` (page 505) 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 554), MultiLineString (page 555), MultiPolygon (page 555), and GeometryCollection (page 556). For details on all supported GeoJSON objects, see [GeoJSON Objects](page 553).
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 497) 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 496) 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.createIndex()` method, specifying the location field as the key and specify the string literal "2dsphere" as the index type:

```
db.collection.createIndex( { <location field> : "2dsphere" } )
```

Unlike a compound `2d` (page 497) 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 `2dsphere` indexes, see *Create a 2dsphere Index* (page 530).

**2d Indexes** Use a `2d` index for data stored as points on a two-dimensional plane. The `2d` index is intended for legacy coordinate pairs used in MongoDB 2.2 and earlier.

Use a `2d` index if:

- your database has legacy location data from MongoDB 2.2 or earlier, and
- you do not intend to store any location data as *GeoJSON* objects.


**Considerations** The `geoNear` command and the `$geoNear` pipeline stage require that a collection have *at most* only one `2d` index and/or only one `2dsphere` index (page 496) whereas *geospatial query operators* (e.g. `$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 496) 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 505) by default and ignores the `sparse: true` (page 505) 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**  
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 496) 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 505) by default and ignore the `sparse: true` (page 505) 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 536). For information and example on querying a haystack index, see Query a Haystack Index (page 536).
2d Index Internals  This document provides a more in-depth explanation of the internals of MongoDB’s 2d geospatial indexes. This material is not necessary for normal operations or application development but may be useful for troubleshooting and for further understanding.

Calculation of Geohash Values for 2d Indexes  When you create a geospatial index on legacy coordinate pairs, MongoDB computes geohash values for the coordinate pairs within the specified location range (page 533) 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.createIndex( { "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:

```javascript
db.records.createIndex( { "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

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 502).

#### Create Text Index

To create a text index, use the `db.collection.createIndex()` 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:

```javascript
db.reviews.createIndex( { 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 539) and [Wildcard Text Indexes](page 500).

#### 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:

```javascript
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 543).

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:

```javascript
db.collection.createIndex( { a: 1, "$**": "text" } )
```
As with all compound text indexes (page 501), 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 501).

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 556).

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 540).

sparse Property

text indexes are sparse (page 505) by default and ignores the sparse: true (page 505) 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 501); 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 501) for additional limitations.

For an example of a compound text index, see Limit the Number of Entries Scanned (page 544).

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 542).
Storage Requirements and Performance Costs

*text* indexes have the following storage requirements and performance costs:

- *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 287).
- *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 545).

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 545).

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 661) a collection using a hashed shard key (page 674). 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 696) 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:

```javascript
db.active.createIndex( { 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 503) The TTL index is used for TTL collections, which expire data after a period of time.

**Unique Indexes** (page 504) A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

**Sparse Indexes** (page 505) A sparse index does not index documents that do not have the indexed field.

### TTL Indexes

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.createIndex()` method with the `expireAfterSeconds` option on a field whose value is either a `date` (page 180) or an array that contains `date values` (page 180).

For example, to create a TTL index on the `lastModifiedDate` field of the `eventlog` collection, use the following operation in the `mongo` shell:

```javascript
db.eventlog.createIndex( { "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 73) operations in the output of `db.currentOp()` or in the data collected by the `database profiler` (page 227).
Timing of the Delete Operation When you build a TTL index in the background (page 508), 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 209) because MongoDB cannot remove documents from a capped collection.
- You cannot use createIndex() 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 211).

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field.

To create a unique index, use the db.collection.createIndex() 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.createIndex( { "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:

```
{ a: [ { b: 5 }, { b: 5 } ] }
```

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:

```
{ 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 505) 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 502).

See also:

Create a Unique Index (page 519)

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field, even if the index field contains a null value. The index skips over any document that is missing the indexed field. The index is “sparse” because it does not include all documents of a collection. By contrast, non-sparse indexes contain all documents in a collection, storing null values for those documents that do not contain the indexed field.

To create a sparse index, use the `db.collection.createIndex()` 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:

```
{ "xmpp_id": 1 }, { sparse: true } }
```

Note: Do not confuse sparse indexes in MongoDB with block-level indexes in other databases. Think of them as dense indexes with a specific filter.

3http://en.wikipedia.org/wiki/Database_index#Sparse_index
Behavior

**sparse Index and Incomplete Results**  Changed in version 2.6.

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 506) for an example that details the behavior.

**Indexes that are sparse by Default** 2dsphere (version 2) (page 496), 2d (page 497), geoHaystack (page 498), and text (page 500) 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 496), 2d (page 497), or geoHaystack (page 498) 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 500) 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 504) 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:

```json
{ "_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`:

```javascript
db.scores.createIndex( { 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:

```javascript
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:

```javascript
{ "_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:

```json
{ "_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`:

```javascript
db.scores.createIndex( { 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:

```javascript
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:

```javascript
{ "_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()`:

```javascript
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:

```javascript
{ "_id" : ObjectId("523b6e6ffb408eea0eec2649"), "userid": "nina", "score" : 90 }
{ "_id" : ObjectId("523b6e61fb408eea0eec2648"), "userid": "abby", "score" : 82 }
```

See also:
`explain()` and *Analyze Query Performance* (page 111)

**Sparse Index with Unique Constraint**

Consider a collection `scores` that contains the following documents:

```javascript
{ "_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 504) and sparse filter on the `score` field using the following operation:

```javascript
db.scores.createIndex( { 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 92):

```javascript
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:

```javascript
db.scores.insert( { "userid": "AAAAAAA", "score": 82 } )
db.scores.insert( { "userid": "BBBBBBB", "score": 90 } )
```

### 8.2.3 Index Creation

MongoDB provides several options that *only* affect the creation of the index. Specify these options in a document as the second argument to the `db.collection.createIndex()` method. This section describes the uses of these
creation options and their behavior.

Related

Some options that you can specify to `createIndex()` options control the properties of the index (page 503), which are not index creation options. For example, the `unique` (page 504) 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 503) 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.createIndex( { 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.createIndex( { 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 `createIndex()` 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.

**Interrupted Index Builds**

If a background index build is in progress when the `mongod` process terminates, when the instance restarts the index build will restart as foreground index build. If the index build encounters any errors, such as a duplicate key error, the `mongod` will exit with an error.

To start the `mongod` after a failed index build, use the `storage.indexBuildRetry` or `--noIndexBuildRetry` to skip the index build on start up. ... _index-creation-building-indexes-on-secondaries:

**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 521) for a complete procedure for building indexes on secondaries.

**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`:

```javascript
db.products.createIndex( { 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**

\(^4\)https://api.mongodb.org/
Issue the following command to create an index on item and quantity and specify inventory as the index name:

```javascript
db.products.createIndex( { 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

New in version 2.6.

MongoDB can use the intersection of multiple indexes to fulfill queries. 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 }`

MongoDB can use the intersection of the two indexes to support the following query:

```javascript
db.orders.find( { item: "abc123", qty: { $gt: 15 } } )
```

To determine if MongoDB used index intersection, run explain(); the results of explain() will include either an AND_SORTED stage or an AND_HASH stage.

### 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:

```javascript
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:

---

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.
The compound index can support the following queries:

```javascript
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:

```javascript
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:

```javascript
{ 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 547)

### 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:

```javascript
{ qty: 1 }
{ status: 1, ord_date: -1 }
{ status: 1 }
{ ord_date: -1 }
```

MongoDB cannot use index intersection for the following query with sort:

```javascript
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.

```javascript
db.orders.find( { qty: { $gt: 10 }, status: "A" } ).sort( { ord_date: -1 } )
```

### 8.2.5 Multikey Index Bounds

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 $[ [ 3, \text{Infinity} ] ]$ and $[ [ -\text{Infinity}, 6 ] ]$, 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 $\text{elemMatch}$ joins the predicates.

For example, a collection `survey` contains documents with a field `item` and an array field `ratings`:

```json
{ _id: 1, item: "ABC", ratings: [ 2, 9 ] }
{ _id: 2, item: "XYZ", ratings: [ 4, 3 ] }
```

Create a multikey index (page 491) on the `ratings` array:

```javascript
db.survey.createIndex( { ratings: 1 } )
```

The following query uses $\text{elemMatch}$ to require that the array contains at least one single element that matches both conditions:

```javascript
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. $\text{gte}: 3$) are $[ [ 3, \text{Infinity} ] ]$;
- the bounds for the less than or equal to 6 predicate (i.e. $\text{lte}: 6$) are $[ [ -\text{Infinity}, 6 ] ]$.

Because the query uses $\text{elemMatch}$ 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 $\text{elemMatch}$, MongoDB cannot intersect the multikey index bounds. Consider the following query:

```javascript
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 $[ [ 3, \text{Infinity} ] ]$ or $[ [ -\text{Infinity}, 6 ] ]$. 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 $[ [ 3, \text{Infinity} ] ]$ and bounds on field `b` of $[ [ -\text{Infinity}, 6 ] ]$, compounding the bounds results in the use of both bounds:

```javascript
{ a: [ [ 3, \text{Infinity} ] ], b: [ [ -\text{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, \text{Infinity} ] ]`. 

---

512 Chapter 8. Indexes
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`:

```json
{ _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:

```javascript
db.survey.createIndex( { item: 1, ratings: 1 } )
```

The following query specifies a condition on both keys of the index:

```javascript
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:

```javascript
{ 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 171) in the index specification. For instance, given the following array of embedded documents:

```json
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`:

```json
{ _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`:

```javascript
db.survey2.createIndex( { "item": 1, "ratings.score": 1, "ratings.by": 1 } )
```

The following query specifies a condition on all three fields:

```javascript
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:

```json
{  
  "item" : [ [ "XYZ", "XYZ" ]],  
  "ratings.score" : [ [ -Infinity, 5 ]],  
  "ratings.by" : [ [ MinKey, MaxKey ]] 
}
```

Or, MongoDB may choose to compound the item bounds with "ratings.by" bounds:

```json
{  
  "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 514) 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 171), 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:

```javascript
db.survey2.createIndex( { "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:

```javascript
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:

```json
{ "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:

```javascript
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":

```javascript
{
    "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:

```javascript
db.survey3.createIndex( { "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:

```javascript
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":

```javascript
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 } },
```
Create a **compound index** (page 489) on the `ratings.score.q1` and the `ratings.score.q2` fields:

```javascript
db.survey4.createIndex( {
    "ratings.score.q1": 1,
    "ratings.score.q2": 1,
    "ratings.certainty.q1": 1,
    "ratings.certainty.q2": 1
} )
```

Consider the following query with two `$elemMatch` clauses:

```javascript
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:
  ```javascript
  { "ratings.score.q1" : [ [ 5, 5 ] ], "ratings.score.q2" : [ [ 5, 5 ] ] }
  ```

- the bounds for the "ratings.certainty" predicate are the compound bounds:
  ```javascript
  { "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":

```javascript
{
    "ratings.score.q1" : [ [ 5, 5 ] ],
    "ratings.score.q2" : [ [ 5, 5 ] ],
    "ratings.certainty.q1" : [ [ MinKey, MaxKey ] ],
    "ratings.certainty.q2" : [ [ MinKey, MaxKey ] ]
}
```

### 8.2.6 Additional Resources

- Quick Reference Cards

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 486) document.

**Index Creation Tutorials** (page 517) Create and configure different types of indexes for different purposes.

**Index Management Tutorials** (page 524) Monitor and assess index performance and rebuild indexes as needed.

**Geospatial Index Tutorials** (page 529) Create indexes that support data stored as GeoJSON objects and legacy coordinate pairs.

**Text Search Tutorials** (page 539) Build and configure indexes that support full-text searches.

**Indexing Strategies** (page 546) 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 517) Build an index for any field on a collection.

**Create a Compound Index** (page 518) Build an index of multiple fields on a collection.

**Create a Unique Index** (page 519) Build an index that enforces unique values for the indexed field or fields.

**Create a Sparse Index** (page 520) 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 521) 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 521) 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 523) 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 523) A \( \{ \_id: 0 \} \) index is necessary if you need to roll back from MongoDB version 2.0 (or later) to MongoDB version 1.8.

Create an Index

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection. 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 518) for instructions on building compound indexes.
Create an Index on a Single Field

To create an index, use `createIndex()` or a similar method from your driver\(^7\). The `createIndex()` 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:

```javascript
db.records.createIndex( { 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:

```javascript
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:

```javascript
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.

**Additional Considerations**

Although indexes can improve query performances, indexes also present some operational considerations. See `Operational Considerations for Indexes` (page 149) 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 508). To build indexes on replica sets, see the `Build Indexes on Replica Sets` (page 521) section for more information.

---

**Note:** To build or rebuild indexes for a replica set see `Build Indexes on Replica Sets` (page 521).

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 518), `Indexing Tutorials` (page 517) and `Index Concepts` (page 486) for more information.

---

**Create a Compound Index**

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a `collection`. MongoDB supports indexes that include content on a single field, as well as `compound indexes` (page 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:

\(^7\)https://api.mongodb.org/
db.collection.createIndex( { 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.createIndex( { 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 508). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 521) section for more information.

**Note:** To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 521).

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 517), *Indexing Tutorials* (page 517) and *Index Concepts* (page 486) for more information.

**Create a Unique Index**

MongoDB allows you to specify a *unique constraint* (page 504) on an index. These constraints prevent applications from inserting *documents* that have duplicate values for the inserted fields.

MongoDB cannot create a *unique index* (page 504) on the specified index field(s) if the collection already contains data that would violate the unique constraint for the index.

**Unique Index on a Single Field**

To create a *unique index* (page 504), consider the following prototype:

```
db.collection.createIndex( { a: 1 }, { unique: true } )
```

For example, you may want to create a unique index on the "tax-id" field of the *accounts* collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.createIndex( { "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.
Unique Compound Index

You can also enforce a unique constraint on *compound indexes* (page 489), as in the following prototype:

```javascript
db.collection.createIndex( { a: 1, b: 1 }, { unique: true } )
```

These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Unique Index and Missing 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* (page 505) 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.

```javascript
db.collection.createIndex( { a: 1 }, { unique: true, sparse: true } )
```

Refer to the `createIndex()` documentation for additional index creation options.

Create a Sparse Index

Sparse indexes are like non-sparse indexes, except that they omit references to documents that do not include the indexed field. For fields that are only present in some documents sparse indexes may provide a significant space savings. See *Sparse Indexes* (page 505) for more information about sparse indexes and their use.

See also:

*Index Concepts* (page 486) and *Indexing Tutorials* (page 517) for more information.

Prototype

To create a *sparse index* (page 505) on a field, use an operation that resembles the following prototype:

```javascript
db.collection.createIndex( { 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.

```javascript
db.users.createIndex( { 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 505) section for more information.
Create a Hashed Index

New in version 2.4.

Hashed indexes (page 502) 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 674) for more information about hashed indexes in sharded clusters, as well as Index Concepts (page 486) and Indexing Tutorials (page 517) for more information about indexes.

Procedure

To create a hashed index (page 502), specify hashed as the value of the index key, as in the following example:

Example

Specify a hashed index on _id

```javascript
db.collection.createIndex( { _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

For replica sets, secondaries will begin building indexes after the primary finishes building the index. In sharded clusters, the mongos will send `createIndex()` 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 517) and Index Concepts (page 486) 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 593) 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.

• Before version 2.6 Backgound index creation operations (page 508) 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. 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 createIndex() in the mongo shell, or comparable method in your driver. This operation will create or rebuild the index on this mongod instance

For example, to create an ascending index on the username field of the records collection, use the following mongo shell operation:

```
db.records.createIndex( { username: 1 } )
```

See also: Create an Index (page 517) and Create a Compound Index (page 518) 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 523). 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 522)
2. Build the Index (page 522)
3. Restart the Program mongod (page 522)

---

8 By running the mongod on a different port, you ensure that the other members of the replica set and all clients will not contact the member while you are building the index.
Build the Index on the Primary  To build an index on the primary you can either:

1. **Build the index in the background** (page 523) 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 522)

   (b) **Build the Index** (page 522)

   (c) **Restart the Program mongod** (page 522)

Building the index on the background, takes longer than the foreground index build and results in a less compact index structure. Additionally, the background index build may impact write performance on the primary. However, building the index in the background allows the set to be continuously up for write operations while MongoDB builds the index.

**Build Indexes in the Background**

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 508) allows read and write operations to continue while building the index.

See also:

*Index Concepts* (page 486) and *Indexing Tutorials* (page 517) 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 `createIndex()` operation, as in the following index:

```
{ a: 1 }, { background: true }
```

Consider the section on **background index construction** (page 508) 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 `createIndex()` 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.createIndex( { name : 1 } , { v : 0 } )
```

**See also:**

*Index Performance Enhancements* (page 910).

### 8.3.2 Index Management Tutorials

Instructions for managing indexes and assessing index performance and use.

**Remove Indexes** *(page 524)* Drop an index from a collection.

**Modify an Index** *(page 525)* Modify an existing index.

**Rebuild Indexes** *(page 526)* In a single operation, drop all indexes on a collection and then rebuild them.

**Manage In-Progress Index Creation** *(page 527)* Check the status of indexing progress, or terminate an ongoing index build.

**Return a List of All Indexes** *(page 527)* Obtain a list of all indexes on a collection or of all indexes on all collections in a database.

**Measure Index Use** *(page 528)* Study query operations and observe index use for your database.

**Remove Indexes**

To remove an index from a collection use the `dropIndex()` method and the following procedure. If you simply need to rebuild indexes you can use the process described in the *Rebuild Indexes* (page 526) document.

**See also:**

*Indexing Tutorials* (page 517) and *Index Concepts* (page 486) 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:

```javascript
db.accounts.dropIndex( { "tax-id": 1 } )
```

The operation returns a document with the status of the operation:

```json
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes before removing this index.

For text (page 500) indexes, pass the index name to the `db.collection.dropIndex()` method. See *Use the Index Name to Drop a text Index* (page 542) 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 `createIndex()` method create a unique index.

```javascript
db.orders.createIndex(  
  { "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 517) and *Index Creation Tutorials* (page 517) for more information on creating indexes.

**Step 2: Attempt to modify the index.**

To modify an existing index, you **cannot** just re-issue the `createIndex()` 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 `createIndex()` method.

```javascript
db.orders.createIndex(  
  { "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.

```javascript
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 524) for more information about dropping indexes.

Step 4: Recreate the index without the unique constraint.

Recreate the index without the unique constraint.

```javascript
db.orders.createIndex(
    { "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 486).

Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method to rebuild all indexes on a collection in a single operation. This operation drops all indexes, including the `_id` index (page 487), and then rebuilds all indexes.

See also:

* Index Concepts (page 486) and Indexing Tutorials (page 517).

Process

The operation takes the following form:

```javascript
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```javascript
{
    "nIndexesWas" : 2,
    "msg" : "indexes dropped for collection",
    "nIndexes" : 2,
    "indexes" : [
        {
            "key" : {
                "_id" : 1,
                "tax-id" : 1
            },
            "ns" : "records.accounts",
            "name" : "_id_"
        }
    ]
}
```
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 521).

**Manage In-Progress Index Creation**

**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 521).

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**

When performing maintenance you may want to check which indexes exist on a collection. In the mongo shell, you can use the `getIndexes()` method to return a list of the indexes on a collection.

**See also:**

*Index Concepts* (page 486) and *Indexing Tutorials* (page 517) 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.

For example, to view all indexes on the `people` collection:

[9] https://api.mongodb.org/
db.people.getIndexes()

List all Indexes for a Database

To list all indexes on all collections in a database, you can use the following operation in the mongo shell:

```javascript
db.getCollectionNames().forEach(function(collection) {
    indexes = db[collection].getIndexes();
    print("Indexes for " + collection + ":");
    printjson(indexes);
});
```

MongoDB 3.0 deprecates direct access to the `system.indexes` collection.

For MongoDB 3.0 deployments using the WiredTiger storage engine, if you run `db.getCollectionNames()` and `db.collection.getIndexes()` from a version of the mongo shell before 3.0 or a version of the driver prior to 3.0 compatible version (page 808), `db.getCollectionNames()` and `db.collection.getIndexes()` will return no data, even if there are existing collections and indexes. For more information, see WiredTiger and Driver Version Compatibility (page 804).

Measure Index Use

Synopsis

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides a number of tools that allow you to study query operations and observe index use for your database.

See also:

Index Concepts (page 486) and Indexing Tutorials (page 517) for more information.

Operations

Return Query Plan with explain() Use the `db.collection.explain()` or the `cursor.explain()` method in executionStats mode to return statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

Run `db.collection.explain()` or the `cursor.explain()` method in allPlansExecution mode to view partial execution statistics collected during plan selection.

db.collection.explain() provides information on the execution of other operations, such as `db.collection.update()`. See `db.collection.explain()` for details.

Control Index Use with hint() To force MongoDB to use a particular index for a `db.collection.find()` operation, specify the index with the `hint()` method. Append the `hint()` method to the `find()` method. Consider the following example:

```javascript
db.people.find(
    { name: "John Doe", zipcode: { $gt: "63000" } }
).hint({ zipcode: 1 })
```

To view the execution statistics for a specific index, append to the `db.collection.find()` the `hint()` method followed by `cursor.explain()`, e.g.:
db.people.find({
    name: "John Doe",
    zipcode: { $gt: "63000" }
}).hint({
    zipcode: 1
}).explain("executionStats")

Or, append `hint()` method to `db.collection.explain().find()`:

db.people.explain("executionStats").find(
    { name: "John Doe",
      zipcode: { $gt: "63000" }
    }).hint({
    zipcode: 1
})

Specify the `$natural` operator to the `hint()` method to prevent MongoDB from using any index:

db.people.find(
    { name: "John Doe",
      zipcode: { $gt: "63000" }
    }).hint({ $natural: 1 })

Instance Index Use Reporting  MongoDB provides a number of metrics of index use and operation that you may want to consider when analyzing index use for your database:

- In the output of `serverStatus`:
  - 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 530)* A 2dsphere index supports data stored as both GeoJSON objects and as legacy coordinate pairs.

*Query a 2dsphere Index (page 531)* 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 533)* Create a 2d index to support queries on data stored as legacy coordinate pairs.

*Query a 2d Index (page 534)* Search for locations using legacy coordinate pairs.

*Create a Haystack Index (page 536)* 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 536)* Search based on location and non-location data within a small area.

*Calculate Distance Using Spherical Geometry (page 537)* Convert distances to radians and back again.
Create a 2dsphere Index

To create a geospatial index for GeoJSON-formatted data, use the `db.collection.createIndex()` method to create a 2dsphere index (page 496). In the index specification document for the `db.collection.createIndex()` method, specify the location field as the index key and specify the string literal "2dsphere" as the value:

```javascript
db.collection.createIndex( { <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 496). 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 553) in a field named `loc`. The coordinate order is longitude, then latitude.

```javascript
db.places.insert({
   loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },
   name: "Central Park",
   category : "Parks"
})
```

```javascript
db.places.insert(
{
   loc : { type: "Point", coordinates: [ -73.88, 40.78 ] },
   name: "La Guardia Airport",
   category : "Airport"
}
)
```

Then, create the 2dsphere (page 496) index.

Create a 2dsphere Index  For example, the following creates a 2dsphere (page 496) index on the location field `loc`:

```javascript
db.places.createIndex( { 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.

```javascript
db.places.createIndex( { loc : "2dsphere" , category : -1, name: 1 } )
```

Unlike the 2d (page 497) index, a compound 2dsphere index does not require the location field to be the first field indexed. For example:

```javascript
db.places.createIndex( { category : 1 , loc : "2dsphere" } )
```
Considerations

Fields with 2dsphere (page 496) 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 497) 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 531).

Query a 2dsphere Index

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:

\[
\text{db.}<\text{collection}>.\text{find( } \{ \text{<location field>:} \}
\]
\[
\{ \text{ $geoIntersects:} \}
\]
\[
\{ \text{ $geometry:} \}
\]
\[
\{ \text{ type: "<GeoJSON object type>"}, \}
\]
\[
\text{coordinates: [ <coordinates> ]}
\]

The following example uses $geoIntersects to select all indexed points and shapes that intersect with the polygon defined by the coordinates array.

\[
\text{db.places.find( } \{ \text{loc:} \}
\]
\[
\{ \text{ $geoIntersects:} \}
\]
\[
\{ \text{ $geometry:} \}
\]
\[
\{ \text{ type: "Polygon"}, \}
\]
\[
\text{coordinates: [ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 1 ], [ 0 , 0 ] ]}
\]

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:

\[
\text{db.}<\text{collection}>.\text{find( } \{ \text{<location field>:} \}
\]
\[
\{ \text{ $near:} \}
\]
\[
\{ \text{ $geometry:} \}
\]
\[
\{ \text{ type: "Point"}, \}
\]
\[
\text{coordinates: [ [ <longitude> , <latitude> ] ]}, \}
\]
\[
\text{$maxDistance: <distance in meters>}
\]

For examples, see $near.

The geoNear command uses the following syntax:

\[
\text{db.runCommand( } \{ \text{geoNear:} <\text{collection}>, \}
\]
\[
\text{near: [ type: "Point", \}
\]
\[
\text{coordinates: [ [ <longitude>, <latitude> ] ]}, \}
\]
\[
\text{spherical: true } \}
\]
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 537).

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 equatorial radius of the earth, 3963.2 miles:

```
db.places.find( { loc : 
    { $geoWithin : 
      { $centerSphere : 
        [ [ -88 , 30 ], 10 / 3963.2 ] }
    } } )
```

Create a 2d Index

To build a geospatial 2d index, use the `createIndex()` method and specify 2d. Use the following syntax:

```
db.<collection>.createIndex( { <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:

```
```
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 the following syntax:

```javascript
db.<collection>.createIndex( {<location field> : "<index type>"} ,
                      { bits : <bit precision> } )
```

For information on the internals of geohash values, see Calculation of Geohash Values for 2d Indexes (page 499).

Query a 2d Index

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:

```javascript
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.

```javascript
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:

```javascript
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 496).

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 537).

Use the following syntax:

```javascript
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 equatorial radius of the earth, 3963.2 miles:

```javascript
db.<collection>.find( { loc : { $geoWithin : { $centerSphere : [ [ 88 , 30 ] , 10 / 3963.2 ] } } } )
```

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:

```javascript
db.<collection>.find( { <location field> : { $near : [ <x> , <y> ] } } )
```

For examples, see $near.

The geoNear command uses the following syntax:

```javascript
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.createIndex( { <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.createIndex( { 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 536).

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 536).

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 537) are not currently supported by haystack indexes.

The `find()` method and `geoNear` command cannot access the haystack index.

---

**Calculate Distance Using Spherical Geometry**

**Note:** While basic queries using spherical distance are supported by the 2d index, consider moving to a 2dsphere index if your data is primarily longitude and latitude.

The 2d index supports queries that calculate distances on a Euclidean plane (flat surface). The index also supports the following query operators and command that calculate distances using spherical geometry:

- `$nearSphere`
- `$centerSphere`
- `$near`
- `geoNear` command with the `{ spherical: true }` option.

**Important:** These three queries use radians for distance. Other query types do not.

For spherical query operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- **distance to radians:** divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- **radians to distance:** multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The equatorial radius of the Earth is approximately 3,963.2 miles or 6,378.1 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:

```javascript
db.places.find( { loc: { $geoWithin: { $centerSphere: [ [ -74, 40.74 ], 100 / 3963.2 ] } } } )
```

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 538).

The following spherical query, returns all documents in the collection `places` within 100 miles from the point `[ -74, 40.74 ]`.

```javascript
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 537) example:

```javascript
db.runCommand( {
    geoNear: "places",
    near: [ -74, 40.74 ],
    spherical: true,
    distanceMultiplier: 3963.2
} )
```

The output of the above operation would resemble the following:

```javascript
{
    // [ ... ]
    "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 539) A text index allows searches on text strings in the index’s specified fields.

Specify a Language for Text Index (page 540) 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 542) Override the text index name limit for long index names.

Control Search Results with Weights (page 543) 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 544) Create an index to support queries that includes $text expressions and equality conditions.

Text Search in the Aggregation Pipeline (page 545) Perform various text search in the aggregation pipeline.

Create a text Index

You can create a text index on the field or fields whose value is a string or an array of string elements. When creating a text index on multiple fields, you can specify the individual fields or you can use wildcard specifier ($**).

Index Specific Fields

The following example creates a text index on the fields subject and content:

db.collection.createIndex(
    {
        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:

```javascript
db.collection.createIndex(
    { "$**": "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 542) for more information.

Specify a Language for Text Index

This tutorial describes how to specify the default language associated with the text index (page 540) and also how to create text indexes for collections that contain documents in different languages (page 540).

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 556) 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`:

```javascript
db.quotes.createIndex(
    { 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 556) 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:

```json
{
  _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.

```javascript
db.quotes.createIndex( { 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`:

```javascript
db.quotes.createIndex( { quote : "text" },
{ language_override: "idioma" } )
```

The documents of the `quotes` collection may specify a language with the `idioma` field:

```json
{ _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**

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`:

```javascript
db.collection.createIndex( 
{ 
    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.createIndex()` method:

```javascript
db.collection.createIndex( 
{ 
    content: "text",
    "users.comments": "text",
    "users.profiles": "text"
},
{ name: "MyTextIndex" }
)
```

**Use the Index Name to Drop a `text` Index**

Whether the `text` (page 500) index has the default name or you specified a name for the `text` (page 500) index, to drop the `text` (page 500) index, pass the index name to the `db.collection.dropIndex()` method.
For example, consider the index created by the following operation:

```javascript
db.collection.createIndex({
  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:

```javascript
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.createIndex()` method.

**Warning:** Choose the weights carefully in order to prevent the need to reindex.

A collection `blog` has the following documents:

```javascript
{ _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 `createIndex()` method. For example, the following command creates an index on three fields and assigns weights to two of the fields:

```javascript
db.blog.createIndex({
  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:

```javascript
{ _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:

```javascript
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`:

```javascript
db.inventory.createIndex(
  { dept: 1,
    description: "text"
  }
)
```

Then, the text search 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`:

```javascript
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 500)

Text Search in the Aggregation Pipeline

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:
• 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.createIndex( { 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.

```javascript
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.

```javascript
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.

```javascript
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 510) of multiple indexes.

The following documents introduce indexing strategies:

Create Indexes to Support Your Queries (page 547) 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 548) 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 550) 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 550) Selectivity is the ability of a query to narrow results using the index. Selectivity allows MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

Create Indexes to Support Your Queries

An index supports a query when the index contains all the fields scanned by the query. The query scans the index and not the collection. Creating indexes that 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:

```javascript
db.products.createIndex( { "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.

```javascript
db.products.createIndex( { "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:
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 \( \{ \text{x: 1, y: 1, z: 1} \} \) index can also support many of the same queries as the following index:

{ x: 1, z: 1 }

Also, \( \{ \text{x: 1, z: 1} \} \) has an additional use. Given the following query:

db.collection.find( { x: 5 } ).sort( { z: 1 } )

The \( \{ \text{x: 1, z: 1} \} \) index supports both the query and the sort operation, while the \( \{ \text{x: 1, y: 1, z: 1} \} \) index only supports the query. For more information on sorting, see Use Indexes to Sort Query Results (page 548).

Starting in version 2.6, MongoDB can use index intersection (page 510) 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 510) for more details.

Use Indexes to Sort Query Results

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 \text{records}:

db.records.createIndex( { 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:

```javascript
db.data.createIndex( { 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.

<table>
<thead>
<tr>
<th>Example</th>
<th>Index Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.data.find().sort( { a: 1 } )</code></td>
<td><code>{ a: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find().sort( { a: -1 } )</code></td>
<td><code>{ a: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find().sort( { a: 1, b: 1 } )</code></td>
<td><code>{ a: 1, b: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find().sort( { a: -1, b: -1 } )</code></td>
<td><code>{ a: 1, b: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find().sort( { a: 1, b: 1, c: 1 } )</code></td>
<td><code>{ a: 1, b: 1, c: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find( { a: { $gt: 4 } } ).sort( { a: 1, b: 1 } )</code></td>
<td><code>{ a: 1, b: 1 }</code></td>
</tr>
</tbody>
</table>

Consider the following example in which the prefix keys of the index appear in both the query predicate and the sort:

```javascript
db.data.find( { a: 5 } ).sort( { b: 1, c: 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:

<table>
<thead>
<tr>
<th>Example</th>
<th>Index Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db.data.find( { a: 5 } ).sort( { b: 1, c: 1 } )</code></td>
<td><code>{ a: 1, b: 1, c: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find( { b: 3, a: 4 } ).sort( { c: 1 } )</code></td>
<td><code>{ a: 1, b: 1, c: 1 }</code></td>
</tr>
<tr>
<td><code>db.data.find( { a: 5, b: { $lt: 3 } } ).sort( { b: 1 } )</code></td>
<td><code>{ a: 1, b: 1 }</code></td>
</tr>
</tbody>
</table>

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`:

```javascript
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

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:

```javascript
> 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 550).

**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:

```javascript
{ _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:

```javascript
{ _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 528).

For a conceptual introduction to indexes in MongoDB see Index Concepts (page 486).
8.4 Indexing Reference

8.4.1 Indexing Methods in the mongo Shell

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db.collection.createIndex()</td>
<td>Builds an index on a collection.</td>
</tr>
<tr>
<td>db.collection.dropIndex()</td>
<td>Removes a specified index on a collection.</td>
</tr>
<tr>
<td>db.collection.dropIndexes()</td>
<td>Removes all indexes on a collection.</td>
</tr>
<tr>
<td>db.collection.getIndexes()</td>
<td>Returns an array of documents that describe the existing indexes on a collection.</td>
</tr>
<tr>
<td>db.collection.getIndexStats()</td>
<td>Renders a human-readable view of the data collected by indexStats which reflects B-tree utilization.</td>
</tr>
<tr>
<td>db.collection.indexStats()</td>
<td>Renders a human-readable view of the data collected by indexStats which reflects B-tree utilization.</td>
</tr>
<tr>
<td>db.collection.reIndex()</td>
<td>Rebuilds all existing indexes on a collection.</td>
</tr>
<tr>
<td>db.collection.totalIndexSize</td>
<td>Reports the total size used by the indexes on a collection. Provides a wrapper around the totalIndexSize field of the collStats output.</td>
</tr>
<tr>
<td>cursor.explain()</td>
<td>Reports on the query execution plan, including index use, for a cursor.</td>
</tr>
<tr>
<td>cursor.hint()</td>
<td>Specifies an exclusive upper index bound for a cursor. For use with cursor.max().</td>
</tr>
<tr>
<td>cursor.max()</td>
<td>Specifies an exclusive upper index bound for a cursor. For use with cursor.hint().</td>
</tr>
<tr>
<td>cursor.min()</td>
<td>Specifies an inclusive lower index bound for a cursor. For use with cursor.hint().</td>
</tr>
<tr>
<td>cursor.snapshot()</td>
<td>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.</td>
</tr>
</tbody>
</table>

8.4.2 Indexing Database Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createIndexes</td>
<td>Builds one or more indexes for a collection.</td>
</tr>
<tr>
<td>dropIndexes</td>
<td>Removes indexes from a collection.</td>
</tr>
<tr>
<td>compact</td>
<td>Defragments a collection and rebuilds the indexes.</td>
</tr>
<tr>
<td>reIndex</td>
<td>Rebuilds all indexes on a collection.</td>
</tr>
<tr>
<td>validate</td>
<td>Internal command that scans for a collection’s data and indexes for correctness.</td>
</tr>
<tr>
<td>indexStats</td>
<td>Experimental command that collects and aggregates statistics on all indexes.</td>
</tr>
<tr>
<td>geoNear</td>
<td>Performs a geospatial query that returns the documents closest to a given point.</td>
</tr>
<tr>
<td>geoSearch</td>
<td>Performs a geospatial query that uses MongoDB’s haystack index functionality.</td>
</tr>
<tr>
<td>geoWalk</td>
<td>An internal command to support geospatial queries.</td>
</tr>
<tr>
<td>checkShardingIndex</td>
<td>Internal command that validates index on shard key.</td>
</tr>
</tbody>
</table>

8.4.3 Geospatial Query Selectors

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$geoWithin</td>
<td>Selects geometries within a bounding GeoJSON geometry (page 553). The 2dsphere (page 496) and 2d (page 497) indexes support $geoWithin.</td>
</tr>
<tr>
<td>$geoIntersects</td>
<td>Selects geometries that intersect with a GeoJSON geometry. The 2dsphere (page 496) index supports $geoIntersects.</td>
</tr>
<tr>
<td>$near</td>
<td>Returns geospatial objects in proximity to a point. Requires a geospatial index. The 2dsphere (page 496) and 2d (page 497) indexes support $near.</td>
</tr>
<tr>
<td>$nearSphere</td>
<td>Returns geospatial objects in proximity to a point on a sphere. Requires a geospatial index. The 2dsphere (page 496) and 2d (page 497) indexes support $nearSphere.</td>
</tr>
</tbody>
</table>
8.4.4 Indexing Query Modifiers

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$explain</td>
<td>Forces MongoDB to report on query execution plans. See explain().</td>
</tr>
<tr>
<td>$hint</td>
<td>Forces MongoDB to use a specific index. See hint().</td>
</tr>
<tr>
<td>$max</td>
<td>Specifies an exclusive upper limit for the index to use in a query. See max().</td>
</tr>
<tr>
<td>$min</td>
<td>Specifies an inclusive lower limit for the index to use in a query. See min().</td>
</tr>
<tr>
<td>$returnKey</td>
<td>Forces the cursor to only return fields included in the index.</td>
</tr>
<tr>
<td>$snapshot</td>
<td>Forces the query to use the index on the _id field. See snapshot().</td>
</tr>
</tbody>
</table>

8.4.5 Other Index References

*GeoJSON Objects* (page 553)  Supported GeoJSON objects.

*Text Search Languages* (page 556)  Supported languages for text indexes (page 500) and $text query operations.

GeoJSON Objects

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\(^\text{10}\):

```
{ type: "Point", coordinates: [ 40, 5 ] }
```

LineString

New in version 2.4.

The following example specifies a GeoJSON LineString\(^\text{11}\):

```
{ type: "LineString", coordinates: [ [ 40, 5 ], [ 41, 6 ] ] }
```

\(^\text{10}\)[http://geojson.org/geojson-spec.html#point]\(^\text{11}\)[http://geojson.org/geojson-spec.html#linestring]
**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 ] ]
  ]
}
```

**MultiPoint**

New in version 2.6: Requires 2dsphere (Version 2) (page 496)

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 ]
  ]
}
```

---

\(^{12}\)http://geojson.org/geojson-spec.html#polygon

\(^{13}\)http://geojson.org/geojson-spec.html#multipoint
MultiLineString

New in version 2.6: Requires 2dsphere (Version 2) (page 496)

The following example specifies a GeoJSON MultiLineString\(^{14}\):

```json
{
  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 ] ]
    ]
}
```

MultiPolygon

New in version 2.6: Requires 2dsphere (Version 2) (page 496)

The following example specifies a GeoJSON MultiPolygon\(^{15}\):

```json
{
  type: "MultiPolygon",
  coordinates:
    [ 
      [ [ -73.958, 40.8003 ], [ -73.9498, 40.7968 ], [ -73.9737, 40.7648 ], [ -73.9814, 40.7681 ] ]
    ]
}
```

\(^{14}\)http://geojson.org/geojson-spec.html#multilinestring

\(^{15}\)http://geojson.org/geojson-spec.html#multipolygon
GeometryCollection

New in version 2.6: Requires 2dsphere (Version 2) (page 496)

The following example stores coordinates of GeoJSON type GeometryCollection:\(^{16}\):

```
{
  type: "GeometryCollection",
  geometries: [
    {
      type: "MultiPoint",
      coordinates: [
        [ -73.958, 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 500), the $text operator supports 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
- nl or dutch
- en or english
- fi or finnish
- fr or french
- de or german
- hu or hungarian
- it or italian

\(^{16}\)http://geojson.org/geojson-spec.html#geometrycollection
• 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.
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 559) An introduction to replica sets, their behavior, operation, and use.

**Replication Concepts** (page 563) The core documentation of replica set operations, configurations, architectures and behaviors.

- *Replica Set Members* (page 564) Introduces the components of replica sets.
- *Replica Set Deployment Architectures* (page 572) Introduces architectural considerations related to replica sets deployment planning.
- *Replica Set High Availability* (page 580) Presents the details of the automatic failover and recovery process with replica sets.
- *Replica Set Read and Write Semantics* (page 585) 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 602) Tutorials for common tasks related to the use and maintenance of replica sets.

**Replication Reference** (page 651) Reference for functions and operations related to replica sets.

### 9.1 Replication Introduction

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 564) accepts all write operations from clients. A replica set can have only one primary. To support replication, the primary records all changes to its data sets in its oplog (page 593). For more information on primary node operation, see Replica Set Primary (page 564).

The secondaries (page 567) 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 567).

You may add an extra mongod instance to a replica set as an arbiter (page 571). 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

---

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 \textit{\texttt{\{w: majority\}}} write concern (page 130). The node that can complete \texttt{\{w: majority\}} 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 \texttt{primary} (page 657).
An arbiter (page 571) will always be an arbiter whereas a primary (page 564) may step down and become a secondary (page 567) and a secondary (page 567) 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 593) and Replica Set Data Synchronization (page 594).

**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 580) and Rollbacks During Replica Set Failover (page 584) for more information.
Read Operations

When a replica set has one and only one primary, reads from that primary provide strict consistency.  

By default, clients read from the primary; however, clients can specify a read preference (page 588) to send read operations to secondaries. Asynchronous replication (page 561) 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 588).

In MongoDB, clients can see the results of writes before they are made durable:

• Regardless of write concern (page 130), 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 584).

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 579), or control the outcome of elections by adjusting the priority of some members. Replica sets also support dedicated members for reporting, disaster recovery, or backup functions.

See Priority 0 Replica Set Members (page 567), Hidden Replica Set Members (page 569) and Delayed Replica Set Members (page 570) for more information.

9.1.3 Additional Resources

• Quick Reference Cards  

• Webinar: Managing Your Mission Critical App - Ensuring Zero Downtime

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 559). For documentation of the administration of replica sets, see Replica Set Tutorials (page 602). The Replication Reference (page 651) documents commands and operations specific to replica sets.

Replica Set Members (page 564) Introduces the components of replica sets.

Replica Set Primary (page 564) The primary is the only member of a replica set that accepts write operations.

Replica Set Secondary Members (page 567) 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 567) Priority 0 members are secondaries that cannot become the primary.

Hidden Replica Set Members (page 569) Hidden members are secondaries that are invisible to applications. These members support dedicated workloads, such as reporting or backup.

Replica Set Arbiter (page 571) An arbiter does not maintain a copy of the data set but participate in elections.

Replica Set Deployment Architectures (page 572) Introduces architectural considerations related to replica sets deployment planning.
Replica Set High Availability (page 580) Presents the details of the automatic failover and recovery process with replica sets.

Replica Set Elections (page 580) Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Read Preference (page 588) Applications specify read preference to control how drivers direct read operations to members of the replica set.

Replication Processes (page 593) Mechanics of the replication process and related topics.

Master Slave Replication (page 596) 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 all 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 583) or priority 0 (page 567).

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.

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 ??).

Changed in version 3.0.0: A replica set can have up to 50 members (page 800) but only 7 voting members. In previous versions, replica sets can have up to 12 members.

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 593). 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 588) for details on changing the default read behavior.

The replica set can have at most one primary. If the current primary becomes unavailable, an election determines the new primary. See Replica Set Elections (page 580) 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.

---

4 While replica sets are the recommended solution for production, a replica set can support up to 50 members in total. If your deployment requires more than 50 members, you’ll need to use master-slave (page 596) replication. However, master-slave replication lacks the automatic failover capabilities.

5 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 130). The node that can complete \{w: majority\} (page 130) 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 657).
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 593) 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 588) for more information on how clients direct read operations to replica sets.

A secondary can become a primary. If the current primary becomes unavailable, the replica set holds an election to choose which of the secondaries becomes the new primary.

In the following three-member replica set, the primary becomes unavailable. This triggers an election where one of the remaining secondaries becomes the new primary.

See Replica Set Elections (page 580) 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 567).
- Prevent applications from reading from it, which allows it to run applications that require separation from normal traffic. See Hidden Replica Set Members (page 569).
- Keep a running “historical” snapshot for use in recovery from certain errors, such as unintentionally deleted databases. See Delayed Replica Set Members (page 570).

Priority 0 Replica Set Members

A priority 0 member is a secondary that cannot become primary. Priority 0 members cannot trigger elections. Otherwise these members function as normal secondaries. A priority 0 member maintains a copy of the data set, accepts read operations, and votes in elections. Configure a priority 0 member to prevent secondaries from becoming primary, which is particularly useful in multi-data center deployments.

In a three-member replica set, in one data center hosts the primary and a secondary. A second data center hosts one priority 0 member that cannot become primary.
Election for New Primary

New Primary Elected

Data Center 1

Primary
priority: 1

Secondary
priority: 1

Data Center 2

Secondary
priority: 0
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 579), a priority 0 standby ensures that only qualified members become primary.

A priority 0 standby may also be valuable for some members of a set with different hardware or workload profiles. In these cases, deploy a member with priority 0 so it can’t become primary. Also consider using an hidden member (page 569) for this purpose.

If your set already has seven voting members, also configure the member as non-voting (page 583).

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 621).

Hidden Replica Set Members

A hidden member maintains a copy of the primary’s data set but is invisible to client applications. Hidden members are good for workloads with different usage patterns from the other members in the replica set. Hidden members must always be priority 0 members (page 567) and so cannot become primary. The `db.isMaster()` method does not display hidden members. Hidden members, however, may vote in elections (page 580).

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.

![Hidden Member Diagram]

Behavior

Read Operations  Clients will not distribute reads with the appropriate read preference (page 588) 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 570) 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 184). To configure a hidden member, see *Configure a Hidden Replica Set Member* (page 623).

## Delayed Replica Set Members

Delayed members contain copies of a replica set’s data set. However, a delayed member’s data set reflects an earlier, or delayed, state of the set. For example, if the current time is 09:52 and a member has a delay of an hour, the delayed member has no operation more recent than 08:52.

Because delayed members are a “rolling backup” or a running “historical” snapshot of the data set, they may help you recover from various kinds of human error. For example, a delayed member can make it possible to recover from unsuccessful application upgrades and operator errors including dropped databases and collections.

### Considerations

**Requirements** Delayed members:

- **Must be** *priority 0* (page 567) members. Set the priority to 0 to prevent a delayed member from becoming primary.
- **Should be** *hidden* (page 569) 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 593).

**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* equal to 0, *hidden* equal to true, and its *slaveDelay* equal to the number of seconds of delay:
To configure a delayed member, see *Configure a Delayed Replica Set Member* (page 624).

**Replica Set Arbiter**

An arbiter does not have a copy of data set and cannot become a primary. Replica sets may have arbiters to add a vote in *elections of for primary* (page 580). Arbiters always have exactly 1 vote election, and thus 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 614).

**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.

---

9.2. Replication Concepts
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 342) for more information.

As with all MongoDB components, run arbiters in trusted network environments.

### 9.2.2 Replica Set Deployment Architectures

The architecture of a replica set affects the set’s capacity and capability. This document provides strategies for replica set deployments and describes common architectures.

The standard replica set deployment for production system is a three-member replica set. These sets provide redundancy and fault tolerance. Avoid complexity when possible, but let your application requirements dictate the architecture.

#### 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. Arbiter 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:

<table>
<thead>
<tr>
<th>Number of Members</th>
<th>Majority Required to Elect a New Primary</th>
<th>Fault Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
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 569) or delayed (page 570) 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 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 Tag Sets**

Use replica set tag sets (page 635) to ensure that operations replicate to specific data centers. Tag sets also allow the routing of read operations to specific machines.

See also:

*Data Center Awareness* (page 207) and *Operational Segregation in MongoDB Deployments* (page 208).

**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 574) Three-member replica sets provide the minimum recommended architecture for a replica set.

Replica Sets with Four or More Members (page 576) 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 579) Geographically distributed sets include members in multiple locations to protect against facility-specific failures, such as power outages.

Three Member Replica Sets

The minimum architecture of a replica set has three members. A three member replica set can have either three members that hold data, or two members that hold data and an arbiter.

Primary with Two Secondary Members A replica set with three members that store data has:

- One primary (page 564).
- Two secondary (page 567) members. Both secondaries can become the primary in an election (page 580).

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 580). 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 564).
- One secondary (page 567) member. The secondary can become primary in an election (page 580).
- One arbiter (page 571). The arbiter only votes in elections.

Since the arbiter does not hold a copy of the data, these deployments provides only one complete copy of the data. Arbiters require fewer resources, at the expense of more limited redundancy and fault tolerance.
9.2. Replication Concepts

Election for New Primary

Secondary → Heartbeat ← Secondary

New Primary Elected

Primary → Heartbeat ← Secondary

Primary → Replication ← Secondary

Secondary

Primary

Arbiter (vote only)
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 603).

**Replica Sets with Four or More Members**

**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 50 members, but only 7 voting members. If the replica set already has 7 voting members, additional members must be non-voting members (page 583).

For example, the following 9 member replica set has 7 voting members and 2 non-voting members.

See Non-Voting Members (page 583) 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 567).

For example, the secondary member in the third data center with a priority of 0 cannot become primary:

---

6 While replica sets are the recommended solution for production, a replica set can support up to 50 members in total. If your deployment requires more than 50 members, you’ll need to use master-slave (page 596) replication. However, master-slave replication lacks the automatic failover capabilities.
See also:

Deploy a Replica Set (page 603), Add an Arbiter to Replica Set (page 614), and Add Members to a Replica Set (page 616).

Geographically Distributed Replica Sets

Adding members to a replica set in multiple data centers adds redundancy and provides fault tolerance if one data center is unavailable. Members in additional data centers should have a priority of 0 (page 567) 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 567) 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 567) 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 78), 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 580).

See also:

Deploy a Geographically Redundant Replica Set (page 608).

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments

9.2.3 Replica Set High Availability

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 580) to select a new primary. In some situations, the failover process may require a rollback (page 584).  

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 580) Elections occur when the primary becomes unavailable and the replica set members autonomously select a new primary.

Rollbacks During Replica Set Failover (page 584) A rollback reverts write operations on a former primary when the member rejoins the replica set after a failover.

Replica Set Elections

Replica sets use elections to determine which set member will become primary. Elections occur after initiating a replica set, and also any time the primary becomes unavailable. The primary is the only member in the set that can accept write operations. If a primary becomes unavailable, elections allow the set to recover normal operations without manual intervention. Elections are part of the failover process (page 580).

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.

9Replica sets remove “rollback” data when needed without intervention. Administrators must apply or discard rollback data manually.
9.2. Replication Concepts

Election for New Primary

Secondary \[\xrightarrow{Heartbeat}\] Secondary

New Primary Elected

Primary \[\xrightarrow{Replication}\] Secondary
\[\xrightarrow{Heartbeat}\] Secondary
**Priority Comparisons**  The priority 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 567).

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 567), 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 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 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 579), 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 setting. Non-voting members (page 625) have votes value of 0. All other members have 1 vote.

Changed in version 3.0.0: Members cannot have votes greater than 1. For details, see Replica Set Configuration Validation (page 805).

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 value.

Vetoes in Elections  
All members of a replica set can veto an election, including non-voting members (page 583). 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 567) 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 583) an election and become primary.

Because a replica set can have up to 50 members, but only 7 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 setting equal to 0 in its member configuration:

```json
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "votes" : 0
}
```

10 Remember that hidden (page 569) and delayed (page 570) imply priority 0 (page 567) configuration.
Important: Do not alter the number of votes to control which members will become primary. Instead, modify the priority 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 625).

Rollbacks During Replica Set Failover

A rollback reverts write operations on a former primary when the member rejoins its replica set after a failover. A rollback is necessary only if the primary had accepted write operations that the secondaries had not successfully replicated before the primary stepped down. When the primary rejoins the set as a secondary, it reverts, or “rolls back,” its write operations to maintain database consistency with the other members.

MongoDB attempts to avoid rollbacks, which should be rare. When a rollback does occur, it is often the result of a network partition. Secondaries that can not keep up with the throughput of operations on the former primary, increase the size 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 130) only provides acknowledgment 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 \(w: \text{majority write concern}\) (page 130) 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 130), 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 584).

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 580) and *Replica Set Elections* (page 580).

### 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 564). 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.

Note:

- In MongoDB, clients can see the results of writes before they are made durable:
  - Regardless of write concern (page 130), 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 584).

- *Sharded clusters* where the shards are also replica sets provide the same operational semantics with regards to write and read operations.

---

11 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 \(w: \text{majority write concern}\) (page 130). The node that can complete \(w: \text{majority}\) 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 657).
Write Concern for Replica Sets (page 586) Write concern is the guarantee an application requires from MongoDB to consider a write operation successful.

Read Preference (page 588) Applications specify read preference to control how drivers direct read operations to members of the replica set.

Read Preference Processes (page 591) With replica sets, read operations may have additional semantics and behavior.

Write Concern for Replica Sets

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. “stand-alone”) or a replica set is transparent. However, replica sets offer some configuration options for write.

Verify Write Operations to Replica Sets

For a replica set, the default write concern (page 78) 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.

```javascript
db.products.insert(
  { item: "envelopes", qty : 100, type: "Clasp" },
  { writeConcern: { w: 2, wtimeout: 5000 } }
)
```

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 131).

See also:

Write Method Acknowledgements (page 857)

Modify Default Write Concern

You can modify the default write concern for a replica set by setting the getLastErrorDefaults setting in the replica set configuration (page 652). The following sequence of commands creates a configuration that waits for the write operation to complete on a majority of the voting members before returning:

```javascript
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 584) in the case of replica set failover (page 580).

12 Sharded clusters where the shards are also replica sets provide the same configuration options with regards to write and read operations.
Always ensure that your operations have specified the required write concern for your application.

See also:

*Write Concern* (page 78) and *connections-write-concern*

**Custom Write Concerns**

You can *tag* (page 635) the members of replica sets and use the resulting tag sets to create custom write concerns. See *Configure Replica Set Tag Sets* (page 635) for information on configuring custom write concerns using tag sets.

**Read Preference**

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.\(^\text{13}\)

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.

**Important:** Exercise care when specifying read preferences: Modes other than *primary* (page 657) may return stale data because with *asynchronous replication* (page 561), data in the secondary may not reflect the most recent write operations.\(^\text{1}\)

\(^{13}\) 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 *\(w: \text{majority}\)* write concern (page 130). The node that can complete *\(w: \text{majority}\)* (page 130) 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 657).
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 130), 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 584).

Use Cases

Indications The following are common use cases for using non-primary (page 657) 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 579) and using a non primary read preference or the nearest (page 658). 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 657) if you want an application to read from the primary under normal circumstances, but to allow stale reads from secondaries when the primary is unavailable. This provides a “read-only mode” for your application during a failover.

Counter-Indications In general, do not use secondary (page 658) and secondaryPreferred (page 658) 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.
- 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 become 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 684) active, secondaries may return stale results with missing or duplicated data because of incomplete or terminated chunk migrations.

Sharding (page 661) 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 591) for more information about the internal application of read preferences.

Read Preference Modes

Important: All read preference modes except primary (page 657) may return stale data because secondaries replicate operations from the primary with some delay. Ensure that your application can tolerate stale data if you choose to use a non-primary (page 657) mode.
MongoDB drivers support five read preference modes.

<table>
<thead>
<tr>
<th>Read Preference Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary (page 657)</td>
<td>Default mode. All operations read from the current replica set primary. In most situations, operations read from the primary but if it is unavailable, operations read from secondary members.</td>
</tr>
<tr>
<td>primaryPreferred (page 657)</td>
<td>In most situations, operations read from primary members but if no primary members are available, operations read from the primary.</td>
</tr>
<tr>
<td>secondary (page 658)</td>
<td>All operations read from the secondary members of the replica set.</td>
</tr>
<tr>
<td>secondaryPreferred (page 658)</td>
<td>In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary. Operations read from member of the replica set with the least network latency, irrespective of the member’s type.</td>
</tr>
<tr>
<td>nearest (page 658)</td>
<td>Operations read from member of the replica set with the least network latency, irrespective of the member’s type.</td>
</tr>
</tbody>
</table>

The syntax for specifying the read preference mode is specific to the driver and to the idioms of the host language[^14]. 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 588) and read preference behavior (page 591). See also the documentation for your driver[^15].

Tag Sets

Tag sets allow you to target read operations to specific members of a replica set.

Custom read preferences and write concerns evaluate tag sets in different ways. Read preferences consider the value of a tag when selecting a member to read from. 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 657)
- secondary (page 658)
- secondaryPreferred (page 658)
- nearest (page 658)

Tags are not compatible with mode primary (page 657) and, in general, only apply when selecting (page 591) a secondary member of a set for a read operation. However, the nearest (page 658) 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 591) 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 635) tutorial.

For more information on how read preference modes (page 657) interact with tag sets, see the documentation for each read preference mode (page 656).

[^14]: https://api.mongodb.org/
[^15]: https://api.mongodb.org/
Read Preference Processes

Changed in version 2.2.

MongoDB drivers use the following procedures to direct operations to replica sets and sharded clusters. To determine how to route their operations, applications periodically update their view of the replica set’s state, identifying which members are up or down, which member is primary, and verifying the latency to each mongod instance.

Member Selection

Clients, by way of their drivers, and mongos instances for sharded clusters, periodically update their view of the replica set’s state.

When you select non-primary (page 657) 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 591) 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 592), 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 read preference (page 591) (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 657), 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 657) and tag set (page 590). 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 657).

- After detecting a failover situation, \(^{16}\) the driver attempts to refresh the state of the replica set as quickly as possible.

Changed in version 3.0.0: *mongos* instances take a slightly different approach. *mongos* instances return connections to secondaries to the connection pool after every request. As a result, the *mongos* reevaluates read preference for every operation.

Read Preference in Sharded Clusters

Changed in version 2.2: Before version 2.2, *mongos* did not support the read preference mode semantics (page 657).

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 657), 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.

---

\(^{16}\) 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.
• All nearest (page 658) 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 593), which is a capped collection.

Replica Set Oplog (page 593) The oplog records all operations that modify the data in the replica set.

Replica Set Data Synchronization (page 594) Secondaries must replicate all changes accepted by the primary. This process is the basis of replica set operations.

Replica Set Oplog

The oplog (operations log) is a special capped collection that keeps a rolling record of all operations that modify the data stored in your databases. MongoDB applies database operations on the primary and then records the operations on the primary’s oplog. The secondary members then copy and apply these operations in an asynchronous process. All replica set members contain a copy of the oplog, in the local.oplog.rs (page 654) 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 628) 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 updates that do not increase the size of the documents, 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 649).

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 647) for more information.

**Replica Set Data Synchronization**

In order to maintain up-to-date copies of the shared data set, members of a replica set sync or replicate data from other members. MongoDB uses two forms of data synchronization: *initial sync* (page 594) 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:

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. The clone process only copies valid data, omitting invalid documents.
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`.

Changed in version 3.0: When the clone process omits an invalid document from the sync, MongoDB writes a message to the logs that begins with `Cloner: found corrupt document in <collection>`.

To perform an initial sync, see `Resync a Member of a Replica Set` (page 633).

**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` setting.

Beginning in version 2.2, secondaries avoid syncing from `delayed members` (page 570) and `hidden members` (page 569).

**Validity and Durability**

In a replica set, the set can have at most one primary and only the primary can accept write operations. 

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 130), 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 584).

**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.

---

17 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 130). The node that can complete `{w: majority}` (page 130) 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 657).
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.

9.2.6 Master Slave Replication

**Important:** *Replica sets* (page 563) 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 598) 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 599).

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 654) 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 654) 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 654) specifying the master instance, as in the following operation in the *mongo* shell:

```
1 use local
2 db.sources.find()
3 db.sources.insert({ host: <masterhostname> ,only: databasename } );
```
In line 1, you switch context to the `local` database. In line 2, the `find()` operation should return no documents, to ensure that there are no documents in the `sources` collection. Finally, line 3 uses `db.collection.insert()` to insert the source document into the `local.sources` (page 654) collection. The model of the `local.sources` (page 654) document is as follows:

`host`

The host field specifies the master `mongod` instance, and holds a resolvable hostname, i.e. IP address, or a name from a `host` file, or preferably a fully qualified domain name.

You can append `:<port>` to the host name if the `mongod` is not running on the default 27017 port.

`only`

Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

**Operational Considerations for Replication with Master Slave Deployments**

Master instances store operations in an `oplog` which is a `capped collection` (page 209). 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:

```javascript
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:

```javascript
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({ repl: 1 })

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 317) 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> 18 provide replication that is roughly equivalent to a two-instance master-slave deployment:

```json
{
    _id : 'setName',
    members : [
        { _id : 0, host : "<master>", priority : 1 },
        { _id : 1, host : "<slave>", priority : 0, votes : 0 }
    ]
}
```

See Replica Set Configuration (page 652) for more information about replica set configurations.

18 In replica set configurations, the host field must hold a resolvable hostname.
Convert a Master-Slave Deployment to a Replica Set

To convert a master-slave deployment to a replica set, restart the current master as a one-member replica set. Then remove the data 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:
   ```
   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 614) tutorial to deploy your replica set, picking up from the *Expand the Replica Set* (page 615) section.

Failing over to a Slave (Promotion)

To permanently failover from an 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:

```plaintext
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 654) 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 654) collection, with the following operation sequence:

```plaintext
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 654) 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 602) Instructions for deploying replica sets, as well as adding and removing members from an existing replica set.

Deploy a Replica Set (page 603) Configure a three-member replica set for production systems.

Convert a Standalone to a Replica Set (page 614) Convert an existing standalone mongod instance into a three-member replica set.

Add Members to a Replica Set (page 616) Add a new member to an existing replica set.

Remove Members from Replica Set (page 618) Remove a member from a replica set.

Continue reading from Replica Set Deployment Tutorials (page 602) for additional tutorials of related to setting up replica set deployments.

Member Configuration Tutorials (page 620) Tutorials that describe the process for configuring replica set members.

Adjust Priority for Replica Set Member (page 620) Change the precedence given to a replica set members in an election for primary.

Prevent Secondary from Becoming Primary (page 621) Make a secondary member ineligible for election as primary.

Configure a Hidden Replica Set Member (page 623) 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 620) for more tutorials that describe replica set configuration.

Replica Set Maintenance Tutorials (page 628) Procedures and tasks for common operations on active replica set deployments.

Change the Size of the Oplog (page 628) 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 633) 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 631) Force a replica set member to become primary.

Change Hostnames in a Replica Set (page 642) Update the replica set configuration to reflect changes in members’ hostnames.

Continue reading from Replica Set Maintenance Tutorials (page 628) for descriptions of additional replica set maintenance procedures.

Troubleshoot Replica Sets (page 646) Describes common issues and operational challenges for replica sets. For additional diagnostic information, see FAQ: MongoDB Diagnostics (page 779).

9.3.1 Replica Set Deployment Tutorials

The following tutorials provide information in deploying replica sets.
See also:

*Security Deployment Tutorials* (page 353) for additional related tutorials.

*Deploy a Replica Set* (page 603) Configure a three-member replica set for production systems.

*Deploy a Replica Set for Testing and Development* (page 605) Configure a three-member replica set for either development or testing systems.

*Deploy a Geographically Redundant Replica Set* (page 608) 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 614) Add an arbiter to give a replica set an odd number of voting members to prevent election ties.

*Convert a Standalone to a Replica Set* (page 614) Convert an existing standalone *mongod* instance into a three-member replica set.

*Add Members to a Replica Set* (page 616) Add a new member to an existing replica set.

*Remove Members from Replica Set* (page 618) Remove a member from a replica set.

*Replace a Replica Set Member* (page 620) Update the replica set configuration when the hostname of a member’s corresponding *mongod* instance has changed.

### Deploy a Replica Set

This tutorial describes how to create a three-member *replica set* from three existing *mongod* instances running with *access control* (page 324) disabled.

To deploy a replica set with enabled *access control* (page 324), see *Deploy Replica Set and Configure Authentication and Authorization* (page 353). If you wish to deploy a replica set from a single MongoDB instance, see *Convert a Standalone to a Replica Set* (page 614). For more information on replica set deployments, see the *Replication* (page 559) and *Replica Set Deployment Architectures* (page 572) 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 580) will proceed smoothly. For more about designing replica sets, see *the Replication overview* (page 559).

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 648).
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 572) 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 system’s /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:

```bash
mongo
```

Step 3: Initiate the replica set. Use `rs.initiate()` on the replica set member:

```bash
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 652):

```bash
rs.conf()
```

The replica set configuration object resembles the following:

```json
{
   "_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:

```bash
rs.add("mongodb1.example.net")
r.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:

```bash
rs.status()
```

See also:

*Deploy Replica Set and Configure Authentication and Authorization* (page 353)

**Deploy a Replica Set for Testing and Development**

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 603) tutorial.

This tutorial describes how to create a three-member *replica set* from three existing mongod instances running with *access control* (page 324) disabled.

To deploy a replica set with enabled *access control* (page 324), see *Deploy Replica Set and Configure Authentication and Authorization* (page 353). If you wish to deploy a replica set from a single MongoDB instance, see *Convert
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 580) will proceed smoothly. For more about designing replica sets, see the Replication overview (page 559).

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 648).

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 652) 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\(^{19}\) as an example of a basic automatically-configured replica set.

Refer to *Replica Set Read and Write Semantics* (page 585) for a detailed explanation of read and write semantics in MongoDB.

### Deploy a Geographically Redundant Replica Set

**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 559) and *Replica Set Deployment Architectures* (page 572). For related tutorials, see *Deploy a Replica Set* (page 603) and *Add Members to a Replica Set* (page 616).

**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 583) are within a primary facility, “Site A”. This includes *priority 0 members* (page 567) and *arbiters* (page 571). 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 573) 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 571) 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.

\(^{19}\)https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py
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 572) 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 652):

```
rconf()
```

The replica set configuration object resembles the following:

```json
{
   "_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:

```javascript
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 7: Check the status of the replica set.** Use the `rs.status()` operation:

```javascript
rs.status()
```

**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 567).

1. View the replica set configuration to determine the `members` array position for the member. Keep in mind the array position is not the same as the `_id`:

   ```javascript
   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.

   ```javascript
   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.

**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 567) in Site B, and an arbiter in Site A.

  - Two members in Site A, two priority 0 members (page 567) 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 652):

```
rs.conf()
```

The replica set configuration object resembles the following:

```json
{
    "_id" : "rs0",
    "version" : 1,
    "members" : [ 
        {
            "_id" : 1,
            "host" : "mongodb0.example.net:27017"
        }
    ]
}
```

Step 8: Check the status of the replica set. Use the rs.status() operation:

```
rstatus()
```

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 567).

1. View the replica set configuration to determine the members array position for the member. Keep in mind the array position is not the same as the _id:
   ```
   rs.conf()
   ```

2. Copy the replica set configuration object to a variable (to cfg in the example below). Then, in the variable, set the correct priority for the member. Then pass the variable to `rs.reconfig()` to update the replica set configuration.

   For example, to set priority for the third member in the array (i.e., the member at position 2), issue the following sequence of commands:
   ```
   cfg = rs.conf()
   cfg.members[2].priority = 0
   rs.reconfig(cfg)
   ```

   **Note:** The `rs.reconfig()` shell method can force the current primary to step down, causing an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While most elections complete within a minute, always make sure any replica configuration changes occur during scheduled maintenance periods.

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

**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 611)). 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 609). Ensure that a single facility, “Site A” always has a majority of the members of the set. For example, if a set has five members, deploy three members within Site A and two members in other facilities.
- If you have a majority of the members of the set outside of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.
Add an Arbiter to Replica Set

Arbiters are MongoDB instances that are part of a replica set but do not hold data. Arbiters participate in elections (page 580) 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 MongoDB process is added to the replica set, the arbiter will act like any other MongoDB 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`
- `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 MongoDB 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**

This tutorial describes the process for converting a standalone MongoDB 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 MongoDB instance, see Deploy a Replica Set (page 603).
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/configuration-options](http://docs.mongodb.org/manual/reference/configuration-options) 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 616).

**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

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 572) document.

Maximum Voting Members A replica set can have a maximum of seven voting members (page 580). 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 583) or remove a vote from an existing member.

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 243) 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 603) 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 593). 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 633).

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 572) 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 652):

   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 document. The document must be in the form of a replSetGetConfig.members document. These documents define a replica set member in the same form as the replica set configuration document (page 652).

Important: Specify a value for the _id field of the members document. MongoDB does not automatically populate
the _id field in this case. Finally, the members document must declare the host value. All other fields are optional.

### Example
To add a member with the following configuration:

- an _id of 1.
- a hostname and port number of mongodb3.example.net:27017.
- a priority value within the replica set of 0.
- a configuration as hidden.

Issue the following:

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

---

### Remove Members from Replica Set

To remove a member of a replica set use either of the following procedures.

#### Remove a Member Using rs.remove()

1. Shut down the mongod instance for the member you wish to remove. To shut down the instance, connect using the mongo shell and the db.shutdownServer() method.
2. Connect to the replica set’s current primary. To determine the current primary, use db.isMaster() while connected to any member of the replica set.
3. Use rs.remove() in either of the following forms to remove the member:

```
rs.remove("mongodb3.example.net:27017")
rs.remove("mongodb3.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 652), 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:
```
Assign the current configuration document to the variable `cfg`:

```javascript
        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:

```javascript
        cfg.members.splice(2,1)
```

6. Overwrite the replica set configuration document with the new configuration by issuing the following:

```javascript
        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:

```javascript
    {
        "_id" : "rs",
        "version" : 8,
        "members" : [
            { 
                "_id" : 0,
                "host" : "mongod_A.example.net:27017"
            },
            { 
                "_id" : 1,
                "host" : "mongod_B.example.net:27017"
            }
        ]
    }
```
Replace a Replica Set Member

If you need to change the hostname of a replica set member without changing the configuration of that member or the set, you can use the operation outlined in this tutorial. For example if you must re-provision systems or rename hosts, you can use this pattern to minimize the scope of that change.

Operation

To change the hostname for a replica set member modify the `host` field. The value of `_id` field will not change when you reconfigure the set.

See `Replica Set Configuration` (page 652) 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 580). 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:

```javascript
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 620) Change the precedence given to a replica set members in an election for primary.

* Prevent Secondary from Becoming Primary (page 621) Make a secondary member ineligible for election as primary.

* Configure a Hidden Replica Set Member (page 623) 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 624) 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 625) 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 626) Convert a secondary to an arbiter.

Adjust Priority for Replica Set Member

Overview

The priority settings of replica set members affect the outcomes of `elections` (page 580) for primary. Use this setting to ensure that some members are more likely to become primary and that others can never become primary.
The value of the member’s `priority` setting determines the member’s priority in elections. The higher the number, the higher the priority.

**Considerations**

To modify priorities, you update the `members` array in the replica configuration object. The array index begins with 0. Do not confuse this index value with the value of the replica set member’s `_id` field in the array.

The value of `priority` can be any floating point (i.e. decimal) number between 0 and 1000. The default value for the `priority` field is 1.

To block a member from seeking election as primary, assign it a priority of 0. *Hidden members* (page 569), *delayed members* (page 570), and *arbiters* (page ??) all have `priority` set to 0.

Adjust priority during a scheduled maintenance window. Reconfiguring priority can force the current primary to step down, leading to an election. Before an election the primary closes all open `client` connections.

**Procedure**

**Step 1:** Copy the replica set configuration to a variable. In the `mongo` shell, use `rs.conf()` to retrieve the replica set configuration and assign it to a variable. For example:

```javascript
cfg = rs.conf()
```

**Step 2:** Change each member’s `priority` value. Change each member’s `priority` value, as configured in the `members` array.

```javascript
cfg.members[0].priority = 0.5
cfg.members[1].priority = 2
cfg.members[2].priority = 2
```

This sequence of operations modifies the value of `cfg` to set the priority for the first three members defined in the `members` array.

**Step 3:** Assign the replica set the new configuration. Use `rs.reconfig()` to apply the new configuration.

```javascript
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**

**Overview**

In a replica set, by default all `secondary` members are eligible to become primary through the election process. You can use the `priority` to affect the outcome of these elections by making some members more likely to become primary and other members less likely or unable to become primary.

Secondaries that cannot become primary are also unable to trigger elections. In all other respects these secondaries are identical to other secondaries.

To prevent a `secondary` member from ever becoming a `primary` in a `failover`, assign the secondary a priority of 0, as described here. For a detailed description of secondary-only members and their purposes, see *Priority 0 Replica Set Members* (page 567).
Considerations

When updating the replica configuration object, access the replica set members in the `members` array with the `array index`. The array index begins with 0. Do not confuse this index value with the value of the `_id` field in each document in the `members` array.

Note: MongoDB does not permit the current `primary` to have a priority of 0. To prevent the current primary from again becoming a primary, you must first step down the current primary using `rs.stepDown()`.

Procedure

This tutorial uses a sample replica set with 5 members.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an `election` (page 580). 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 614) 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 652) 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:

```javascript
cfg = rs.conf()
```

The returned document contains a `members` field which contains an array of member configuration documents, one document for each member of the replica set.

Step 2: Assign priority value of 0. To prevent a secondary member from becoming a primary, update the secondary member’s `priority` to 0.

To assign a priority value to a member of the replica set, access the member configuration document using the array index. In this tutorial, the secondary member to change corresponds to the configuration document found at position 2 of the `members` array.

```javascript
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:

```javascript
rs.reconfig(cfg)
```
Configure a Hidden Replica Set Member

Hidden members are part of a replica set but cannot become primary and are invisible to client applications. Hidden members may vote in elections (page 580). For a more information on hidden members and their uses, see Hidden Replica Set Members (page 569).

Considerations

The most common use of hidden nodes is to support delayed members (page 570). If you only need to prevent a member from becoming primary, configure a priority 0 member (page 567).

If the chainingAllowed setting allows secondary members to sync from other secondaries, MongoDB by default prefers non-hidden members over hidden members when selecting a sync target. MongoDB will only choose hidden members as a last resort. If you want a secondary to sync from a hidden member, use the replSetSyncFrom database command to override the default sync target. See the documentation for replSetSyncFrom before using the command.

See also:

Manage Chained Replication (page 641)

Changed in version 2.0: For sharded clusters running with replica sets before 2.0, if you reconfigured a member as hidden, you had to restart mongos to prevent queries from reaching the hidden member.

Examples

Member Configuration Document  To configure a secondary member as hidden, set its priority value to 0 and set its hidden value to true in its member configuration:

```json
{
   "_id" : <num>,
   "host" : <hostname:port>,
   "priority" : 0,
   "hidden" : true
}
```

Configuration Procedure  The following example hides the secondary member currently at the index 0 in the members array. To configure a hidden member, use the following sequence of operations in a mongo shell connected to the primary, specifying the member to configure by its array index in the members array:

```javascript
rs = rs.conf()
rs.members[0].priority = 0
rs.members[0].hidden = true
rs.reconfig(rs)
```
After re-configuring the set, this secondary member has a priority of 0 so that it cannot become primary and is hidden. The other members in the set will not advertise the hidden member in the isMaster or db.isMaster() output. When updating the replica configuration object, access the replica set members in the members array with the array index. The array index begins with 0. Do not confuse this index value with the value of the _id field in each document in the members array.

**Warning:**

- The rs.reconfig() shell method can force the current primary to step down, which causes an election (page 580). 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 614) 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 580)
- Read Preference (page 588)

**Configure a Delayed Replica Set Member**

To configure a delayed secondary member, set its priority value to 0, its hidden value to true, and its slaveDelay value to the number of seconds to delay.

**Important:** The length of the secondary slaveDelay must fit within the window of the oplog. If the oplog is shorter than the slaveDelay window, the delayed member cannot successfully replicate operations.

When you configure a delayed member, the delay applies both to replication and to the member’s oplog. For details on delayed members and their uses, see Delayed Replica Set Members (page 570).

**Example**

The following example sets a 1-hour delay on a secondary member currently at the index 0 in the members array. To set the delay, issue the following sequence of operations in a mongo shell connected to the primary:

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

After the replica set reconfigures, the delayed secondary member cannot become primary and is hidden from applications. The slaveDelay value delays both replication and the member’s oplog by 3600 seconds (1 hour).

When updating the replica configuration object, access the replica set members in the members array with the array index. The array index begins with 0. Do not confuse this index value with the value of the _id field in each document in the members array.
Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 580). 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 614) to ensure that members can quickly obtain a majority of votes in an election for primary.

Related Documents

- `slaveDelay`
- *Replica Set Reconfiguration*
- *Oplog Size* (page 593)
- *Change the Size of the Oplog* (page 628) tutorial
- *Replica Set Elections* (page 580)

Configure Non-Voting Replica Set Member

Non-voting members allow you to add additional members for read distribution beyond the maximum seven voting members. To configure a member as non-voting, set its `votes` value to `0`.

Example

To disable the ability to vote in elections for the fourth, fifth, and sixth replica set members, use the following command sequence in the `mongo` shell connected to the primary. You identify each replica set member by its array index in the `members` array:

```javascript
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

This sequence gives `0` votes to the fourth, fifth, and sixth members of the set according to the order of the `members` array in the output of `rs.conf()`. This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

When updating the replica configuration object, access the replica set members in the `members` array with the *array index*. The array index begins with `0`. Do **not** confuse this index value with the value of the `_id` field in each document in the `members` array.

Warning:

- The `rs.reconfig()` shell method can force the current primary to step down, which causes an *election* (page 580). 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 614) to ensure that members can quickly obtain a majority of votes in an election for primary.
In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary. Use priority to control which members are more likely to become primary.

Related Documents

- votes
- Replica Set Reconfiguration
- Replica Set Elections (page 580)

Convert a Secondary to an Arbiter

If you have a secondary in a replica set that no longer needs to hold data but that needs to remain in the set to ensure that the set can elect a primary (page 580), 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 626).

- 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 627).

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 628) 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 630) Perform maintenance on a member of a replica set while minimizing downtime.

Force a Member to Become Primary (page 631) Force a replica set member to become primary.

Resync a Member of a Replica Set (page 633) 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 635) 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 638) Reconfigure a replica set when a majority of replica set members are down or unreachable.

Manage Chained Replication (page 641) 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 642) Update the replica set configuration to reflect changes in members’ hostnames.

Configure a Secondary’s Sync Target (page 645) Specify the member that a secondary member synchronizes from.

Change the Size of the Oplog

The oplog exists internally as a capped collection, so you cannot modify its size in the course of normal operations. In most cases the default oplog size (page 593) 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 593). For details how oplog size affects delayed members and affects replication lag, see Delayed Replica Set Members (page 570).

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:

```javascript
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:

```bash
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:

```bash
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:

```javascript
use local
```

In `mongo` shell scripts you can use the following operation to set the `db` object:

```javascript
db = db.getSiblingDB('local')
```

Ensure that the `temp` temporary collection is empty by dropping the collection:

```javascript
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:

```javascript
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:

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

Remove the Existing Oplog Collection
Drop the old `oplog.rs` collection in the `local` database. Use the following command:

```javascript
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 $2 \times 1024 \times 1024 \times 1024$ will create a new oplog that’s 2 gigabytes:

```javascript
db.runCommand( { create: "oplog.rs", capped: true, size: (2 * 1024 * 1024 * 1024) } )
```

Upon success, this command returns the following status:

```json
{ "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:

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

To confirm the entry is in the new oplog, use the following operation:

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

Restart the Member  Restart the mongod as a member of the replica set on its usual port. For example:

```javascript
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

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 236) and changing the size of the oplog (page 628).

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:

```javascript
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:

```bash
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:

```bash
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:

```javascript
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 594). From the `mongo` shell, use the following command to verify that the member has caught up from the `RECOVERING` (page 656) state to the `SECONDARY` (page 655) state.

```javascript
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:

```javascript
rs.stepDown(300)
```

After the primary steps down, the replica set will elect a new primary. See `Replica Set Elections` (page 580) for more information about replica set elections.

**Force a Member to Become Primary**

**Overview**

You can force a `replica set` member to become `primary` by giving it a higher `priority` value than any other member in the set.

Optionally, you also can force a member never to become primary by setting its `priority` value to 0, which means the member can never seek `election` (page 580) as primary. For more information, see `Priority 0 Replica Set Members` (page 567).

For more information on priorities, see `priority`.  

---

**9.3. Replica Set Tutorials** 631
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 580) 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()
for i in range(len(cfg.members)): 
  if i == 2: 
    cfg.members[i].priority = 1 
  else: 
    cfg.members[i].priority = 0.5 
rs.reconfig(cfg)
```

The last statement calls `rs.reconfig()` with the modified configuration document to configure m3.example.net to have a higher `replSetGetConfig.members[n].priority` value than the other mongod instances.

The following sequence of events occur:

- m3.example.net and m2.example.net sync with m1.example.net (typically within 10 seconds).
- m1.example.net sees that it no longer has highest priority and, in most cases, steps down. m1.example.net does not step down if m3.example.net's sync is far behind. In that case, m1.example.net waits until m3.example.net is within 10 seconds of its optime and then steps down. This minimizes the amount of time with no primary following failover.
- The step down forces on election in which m3.example.net becomes primary based on its priority setting.
2. Optionally, if `m3.example.net` is more than 10 seconds behind `m1.example.net`'s optime, and if you don’t need to have a primary designated within 10 seconds, you can force `m1.example.net` to step down by running:

   ```
   db.adminCommand({replSetStepDown: 86400, force: 1})
   ```

   This prevents `m1.example.net` from being primary for 86,400 seconds (24 hours), even if there is no other member that can become primary. When `m3.example.net` catches up with `m1.example.net` it will become primary.

   If you later want to make `m1.example.net` primary again while it waits for `m3.example.net` to catch up, issue the following command to make `m1.example.net` seek election again:

   ```
   rs.freeze()
   ```

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

**Force a Member to be Primary Using Database Commands**  
Changed in version 1.8.

Consider a replica set with the following members:

- `mdb0.example.net` - the current primary.
- `mdb1.example.net` - a secondary.
- `mdb2.example.net` - a secondary.

To force a member to become primary use the following procedure:

1. In a mongo shell, run `rs.status()` to ensure your replica set is running as expected.
2. In a mongo shell connected to the mongod instance running on `mdb2.example.net`, freeze `mdb2.example.net` so that it does not attempt to become primary for 120 seconds:

   ```
   rs.freeze(120)
   ```

3. In a mongo shell connected the mongod running on `mdb0.example.net`, step down this instance that the mongod is not eligible to become primary for 120 seconds:

   ```
   rs.stepDown(120)
   ```

   `mdb1.example.net` becomes primary.

   **Note:** During the transition, there is a short window where the set does not have a primary.

For more information, consider the `rs.freeze()` and `rs.stepDown()` methods that wrap the `replSetFreeze` and `replSetStepDown` commands.

**Resync a Member of a Replica Set**

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 594).

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 634).

• 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 634).

### Procedures

#### Automatically Sync a Member

This procedure relies on MongoDB’s regular process for initial sync (page 594). This will store the current data on the member. For an overview of MongoDB initial sync process, see the Replication Processes (page 593) section.

If the instance has no data, you can simply follow the Add Members to a Replica Set (page 616) or Replace a Replica Set Member (page 620) 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 184) 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

Tag sets let you customize write concern and read preferences for a replica set. MongoDB stores tag sets in the replica set configuration object, which is the document returned by `rs.conf()`, in the `members[n].tags` embedded document.

This section introduces the configuration of tag sets. For an overview on tag sets and their use, see Replica Set Write Concern (page 80) and Tag Sets (page 590).

Differences Between Read Preferences and Write Concerns

Custom read preferences and write concerns evaluate tag sets in different ways:

- Read preferences consider the value of a tag when selecting a member to read from.
- 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:

```json
{ "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:

```json
{ "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:

```json
{ "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:

```json
{
   "_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"
   }

9.3. Replica Set Tutorials 635

MongoDB Documentation, Release 3.0.4
```
You could add tag sets to the members of this replica set with the following command sequence in the mongo shell:

```javascript
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:

```javascript
{
  "_id" : "rs0",
  "version" : 2,
  "members" : [
    {
      "_id" : 0,
      "host" : "mongodb0.example.net:27017",
      "tags" : {
        "dc": "east",
        "use": "production"
      }
    },
    {
      "_id" : 1,
      "host" : "mongodb1.example.net:27017",
      "tags" : {
        "dc": "east",
        "use": "reporting"
      }
    },
    {
      "_id" : 2,
      "host" : "mongodb2.example.net:27017",
      "tags" : {
        "use": "production"
      }
    }
  ]
}
```

**Important:** In tag sets, all tag values must be strings.

## Custom Multi-Datacenter Write Concerns

Given a five member replica set with members in two data centers:

1. a facility VA tagged dc.va
2. a facility GTO tagged dc.gto

Create a custom write concern to require confirmation from two data centers using replica set tags, using the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration JavaScript object `conf`:
conf = rs.conf()

2. Add tags to the replica set members reflecting their locations:

   conf.members[0].tags = { "dc.va": "rack1"}
   conf.members[1].tags = { "dc.va": "rack2"}
   conf.members[2].tags = { "dc.gto": "rack1"}
   conf.members[3].tags = { "dc.gto": "rack2"}
   conf.members[4].tags = { "dc.va": "rack1"}
   rs.reconfig(conf)

3. Create a custom getLastErrorModes setting to ensure that the write operation will propagate to at least one member of each facility:

   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 1, "dc.gto": 1 } } }

4. Reconfigure the replica set using the modified conf configuration object:

   rs.reconfig(conf)

To ensure that a write operation propagates to at least one member of the set in both data centers, use the MultipleDC write concern mode as follows:

   db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )

Alternatively, if you want to ensure that each write operation propagates to at least 2 racks in each facility, reconfigure the replica set as follows in the mongo shell:

1. Create a replica set configuration object conf:

   conf = rs.conf()

2. Redefine the getLastErrorModes value to require two different values of both dc.va and dc.gto:

   conf.settings = { getLastErrorModes: { MultipleDC : { "dc.va": 2, "dc.gto": 2}}

3. Reconfigure the replica set using the modified conf configuration object:

   rs.reconfig(conf)

Now, the following write operation will only return after the write operation propagates to at least two different racks in the each facility:

   db.users.insert( { id: "xyz", status: "A" }, { writeConcern: { w: "MultipleDC" } } )

Changed in version 2.6: A new protocol for write operations (page 852) 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: 20

---

20 Since read preferences and write concerns use the value of fields in tag sets differently, larger deployments may have some redundancy.
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:

```json
{ disk: "ssd" }
```

However, to create comparable write concern modes, you would specify a different set of `getLastErrorModes` configuration. Consider the following sequence of operations in the `mongo` shell:

1. Create a replica set configuration object `conf`:

   ```javascript
   conf = rs.conf()
   ```

2. Redefine the `getLastErrorModes` value to configure two write concern modes:

   ```javascript
   conf.settings = {
     "getLastErrorModes" : {
       "ssd" : {
         "ssd" : 1
       },
       "MultipleDC" : {
         "dc.va" : 1,
         "dc.gto" : 1
       }
     }
   }
   ```

3. Reconfigure the replica set using the modified `conf` configuration object:

   ```javascript
   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.

```javascript
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

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 639)
- **Reconfigure by Replacing the Replica Set** (page 639)

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 580) for more information on this situation.
Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of replica set members are down or unreachable. You connect to any surviving member and use the force option to the rs.reconfig() method.

The force option forces a new configuration onto the member. Use this procedure only to recover from catastrophic interruptions. Do not use force every time you reconfigure. Also, do not use the force option in any automatic scripts and do not use force when there is still a primary.

To force reconfiguration:

1. Back up a surviving member.

2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

   ```
   cfg = rs.conf()
   printjson(cfg)
   ```

3. On the same member, remove the down and unreachable members of the replica set from the members array by setting the array equal to the surviving members alone. Consider the following example, which uses the cfg variable created in the previous step:

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

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

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

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

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

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

Reconfigure by Replacing the Replica Set

Use the following procedure only for versions of MongoDB prior to version 2.0. If you’re running MongoDB 2.0 or later, use the above procedure, Reconfigure by Forcing the Reconfiguration (page 639).

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 633)
Manage Chained Replication

Starting in version 2.0, MongoDB supports chained replication. A chained replication occurs when a secondary member replicates from another secondary member instead of from the primary. This might be the case, for example, if a secondary selects its replication target based on ping time and if the closest member is another secondary.

Chained replication can reduce load on the primary. But chained replication can also result in increased replication lag, depending on the topology of the network.

New in version 2.2.2.

You can use the chainingAllowed setting in Replica Set Configuration (page 652) to disable chained replication for situations where chained replication is causing lag.

MongoDB enables chained replication by default. This procedure describes how to disable it and how to re-enable it.

Note: If chained replication is disabled, you still can use replSetSyncFrom to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Disable Chained Replication

To disable chained replication, set the chainingAllowed field in Replica Set Configuration (page 652) to false.

You can use the following sequence of commands to set chainingAllowed to false:

1. Copy the configuration settings into the cfg object:
   
   ```
   cfg = rs.config()
   ```

2. Take note of whether the current configuration settings contain the settings embedded document. If they do, skip this step.

   Warning: To avoid data loss, skip this step if the configuration settings contain the settings embedded document.

   If the current configuration settings do not contain the settings embedded document, create the embedded document by issuing the following command:

   ```
   cfg.settings = {}
   ```

3. Issue the following sequence of commands to set chainingAllowed to false:

   ```
   cfg.settings.chainingAllowed = false
   rs.reconfig(cfg)
   ```

Re-enable Chained Replication

To re-enable chained replication, set chainingAllowed to true. You can use the following sequence of commands:

```
cfg = rs.config()
cfg.settings.chainingAllowed = true
rs.reconfig(cfg)
```
Change Hostnames in a Replica Set

For most replica sets, the hostnames in the host field never change. However, if organizational needs change, you might need to migrate some or all host names.

Note: Always use resolvable hostnames for the value of the host field in the replica set configuration to avoid confusion and complexity.

Overview

This document provides two separate procedures for changing the hostnames in the host field. Use either of the following approaches:

- Change hostnames without disrupting availability (page 643). 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 644). 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 603), and Add Members to a Replica Set (page 616).

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:

```json
{
   "_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 642).

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 652) 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 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 652) 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:

```json
{
  "_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 642).

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**

**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 594) and Manage Chained Replication (page 641) for more information.

For some deployments, implementing a custom replication sync topology may be more effective than the default sync target selection logic. MongoDB provides the ability to specify a host to use as a sync target.

To override the default sync target selection logic, you may manually configure a secondary member’s sync target to temporarily pull oplog entries. The following provide access to this functionality:

  - replSetSyncFrom command, or
  - rs.syncFrom() helper in the mongo shell

**Considerations**

**Sync Logic** Only modify the default sync logic as needed, and always exercise caution. rs.syncFrom() will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run rs.syncFrom() operation before initial sync.
If you run `rs.syncFrom()` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

**Persistence**  
`replSetSyncFrom` and `rs.syncFrom()` provide a temporary override of default behavior. `mongod` will revert to the default sync behavior in the following situations:

- The `mongod` instance restarts.
- The connection between the `mongod` and the sync target closes.

Changed in version 2.4: The sync target falls more than 30 seconds behind another member of the replica set; the `mongod` will revert to the default sync target.

**Target**  
The member to sync from must be a valid source for data in the set. To sync from a member, the member must:

- Have data. It cannot be an arbiter, in startup or recovering mode, and must be able to answer data queries.
- Be accessible.
- Be a member of the same set in the replica set configuration.
- Build indexes with the `buildIndexes` setting.
- A different member of the set, to prevent syncing from itself.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, `mongod` will log a warning but will still replicate from the lagging member.

If you run `replSetSyncFrom` during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

**Procedure**

To use the `replSetSyncFrom` command in the `mongo` shell:

```javascript
db.adminCommand( { replSetSyncFrom: "hostname<:port>" } );
```

To use the `rs.syncFrom()` helper in the `mongo` shell:

```javascript
rs.syncFrom("hostname<:port>");
```

### 9.3.4 Troubleshoot Replica Sets

This section describes common strategies for troubleshooting replica set deployments.

**Check Replica Set Status**

To display the current state of the replica set and current state of each member, run the `rs.status()` method in a mongo shell connected to the replica set’s `primary`. For descriptions of the information displayed by `rs.status()`, see [http://docs.mongodb.org/manual/reference/command/replSetGetStatus](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 570) may show as 0 seconds behind the primary when the inactivity period on the primary is greater than the `slaveDelay` value.

  **Note:** The `rs.status()` method is a wrapper around the `replSetGetStatus` database command.

- Monitor the rate of replication by watching the oplog time in the “replica” graph in the MongoDB Cloud Manager\(^2\) and in Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced\(^2\). For more information see the MongoDB Cloud Manager documentation\(^3\) and Ops Manager documentation\(^4\).

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 78) to require confirmation of replication to secondaries, as described in replica set write concern (page 80). This prevents write operations from returning if replication cannot keep up with the write load.

---

\(^2\)[https://cloud.mongodb.com/?jmp=docs]
\(^2\)[https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs]
\(^2\)[https://docs.cloud.mongodb.com/]
\(^4\)[https://docs.opsmanager.mongodb.com/current]
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 78), 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 78) after every 100, 1,000, or another interval to provide an opportunity for secondaries to catch up with the primary.

For more information see:

- *Replica Acknowledge Write Concern* (page 80)
- *Replica Set Write Concern* (page 84)
- *Oplog Size* (page 593)

**Test Connections Between all Members**

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both “directions.” Networking topologies and firewall configurations can prevent normal and required connectivity, which can block replication.

Consider the following example of a bidirectional test of networking:

**Example**

Given a replica set with three members running on three separate hosts:

- m1.example.net
- m2.example.net
- m3.example.net

1. Test the connection from m1.example.net to the other hosts with the following operation set from m1.example.net:
   
   ```
   mongo --host m2.example.net --port 27017
   mongo --host m3.example.net --port 27017
   ```

2. Test the connection from m2.example.net to the other two hosts with the following operation set from m2.example.net, as in:

   ```
   mongo --host m1.example.net --port 27017
   mongo --host m3.example.net --port 27017
   ```

   You have now tested the connection between m2.example.net and m1.example.net in both directions.

3. Test the connection from m3.example.net to the other two hosts with the following operation set from the m3.example.net host, as in:

   ```
   mongo --host m1.example.net --port 27017
   mongo --host m2.example.net --port 27017
   ```
If any connection, in any direction fails, check your networking and firewall configuration and reconfigure your environment to allow these connections.

**Socket Exceptions when Rebooting More than One Secondary**

When you reboot members of a replica set, ensure that the set is able to elect a primary during the maintenance. This means ensuring that a majority of the set’s ‘votes’ are available.

When a set’s active members can no longer form a majority, the set’s primary steps down and becomes a secondary. The former primary closes all open connections to client applications. Clients attempting to write to the former primary receive socket exceptions and Connection reset errors until the set can elect a primary.

**Example**

Given a three-member replica set where every member has one vote, the set can elect a primary only as long as two members can connect to each other. If two you reboot the two secondaries once, the primary steps down and becomes a secondary. Until the at least one secondary becomes available, the set has no primary and cannot elect a new primary.

For more information on votes, see *Replica Set Elections* (page 580). For related information on connection errors, see *Does TCP keepalive time affect MongoDB Deployments?* (page 779).

**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 593),
- *Delayed Replica Set Members* (page 570), and
- *Check the Replication Lag* (page 647).

**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 628) tutorial.
Oplog Entry Timestamp Error

Consider the following error in mongod output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection. Terminating mongod after 30 <timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the ts field in the last oplog entry causes this error. The correct data type is Timestamp.

Check the type of the ts value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the ts value is a Timestamp. The $type operator allows you to select BSON type 17, is the Timestamp data type.

If the queries don’t return the same document, then the last document in the oplog has the wrong data type in the ts field.

**Example**

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where ts has the Timestamp type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the ts field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
    { $set: { ts: new Timestamp(1347982456000, 1) } })
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

**Duplicate Key Error on local.slaves**

Changed in version 3.0.0.

MongoDB 3.0.0 removes the local.slaves (page 654) collection. For local.slaves error in earlier versions of MongoDB, refer to the appropriate version of the MongoDB Manual.
9.4 Replication Reference

9.4.1 Replication Methods in the \texttt{mongo} Shell

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs.add()</td>
<td>Adds a member to a replica set.</td>
</tr>
<tr>
<td>rs.addArb()</td>
<td>Adds an \textit{arbiter} to a replica set.</td>
</tr>
<tr>
<td>rs.conf()</td>
<td>Returns the replica set configuration document.</td>
</tr>
<tr>
<td>rs.freeze()</td>
<td>Prevents the current member from seeking election as primary for a period of time.</td>
</tr>
<tr>
<td>rs.help()</td>
<td>Returns basic help text for \textit{replica set} functions.</td>
</tr>
<tr>
<td>rs.initiate()</td>
<td>Initializes a new replica set.</td>
</tr>
<tr>
<td>rs.printReplicationInfo()</td>
<td>Prints a report of the status of the replica set from the perspective of the primary.</td>
</tr>
<tr>
<td>rs.printSlaveReplicationInfo()</td>
<td>Re-configures a replica set by applying a new replica set configuration object.</td>
</tr>
<tr>
<td>rs.remove()</td>
<td>Remove a member from a replica set.</td>
</tr>
<tr>
<td>rs.slaveOk()</td>
<td>Sets the \textit{slaveOk} property for the current connection. Deprecated. Use \texttt{readPref()} and \texttt{Mongo.setReadPref()} to set \textit{read preference}.</td>
</tr>
<tr>
<td>rs.status()</td>
<td>Returns a document with information about the state of the replica set.</td>
</tr>
<tr>
<td>rs.stepDown()</td>
<td>Causes the current \textit{primary} to become a secondary which forces an \textit{election}.</td>
</tr>
<tr>
<td>rs.syncFrom()</td>
<td>Sets the member that this replica set member will sync from, overriding the default sync target selection logic.</td>
</tr>
</tbody>
</table>

9.4.2 Replication Database Commands

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>replSetFreeze</td>
<td>Prevents the current member from seeking election as \textit{primary} for a period of time.</td>
</tr>
<tr>
<td>replSetGetStatus</td>
<td>Returns a document that reports on the status of the replica set.</td>
</tr>
<tr>
<td>replSetInitiate</td>
<td>Initializes a new replica set.</td>
</tr>
<tr>
<td>replSetMaintenance</td>
<td>Enables or disables a maintenance mode, which puts a \textit{secondary} node in a RECOVERING state.</td>
</tr>
<tr>
<td>replSetReconfig</td>
<td>Applies a new configuration to an existing replica set.</td>
</tr>
<tr>
<td>replSetStepDown</td>
<td>Forces the current \textit{primary} to \textit{step down} and become a \textit{secondary}, forcing an election.</td>
</tr>
<tr>
<td>replSetSyncFrom</td>
<td>Explicitly override the default logic for selecting a member to replicate from.</td>
</tr>
<tr>
<td>resync</td>
<td>Forces a \texttt{mongod} to re-synchronize from the \textit{master}. For master-slave replication only.</td>
</tr>
<tr>
<td>applyOps</td>
<td>Internal command that applies \textit{oplog} entries to the current data set.</td>
</tr>
<tr>
<td>isMaster</td>
<td>Displays information about this member’s role in the replica set, including whether it is the master.</td>
</tr>
<tr>
<td>replSetGetConfig</td>
<td>Returns the replica set’s configuration object.</td>
</tr>
</tbody>
</table>

9.4.3 Replica Set Reference Documentation

\textit{Replica Set Configuration} (page 652) Complete documentation of the \textit{replica set} configuration object returned by \texttt{rs.conf}().

\textit{The local Database} (page 652) Complete documentation of the content of the \textit{local} database that \texttt{mongod} instances use to support replication.

\textit{Replica Set Member States} (page 654) Reference for the replica set member states.

\textit{Read Preference Reference} (page 656) Complete documentation of the five read preference modes that the MongoDB drivers support.
Replica Set Configuration

You can access the configuration of a replica set using the `rs.conf()` method or the `replSetGetConfig` command.

Example Output

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>  
    }  
  }
```

For a description of the replica set configuration settings, see `replSetGetConfig-output`.

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

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` with diagnostic information about the `mongod` instance itself and host information. `startup_log` is a capped collection. This information is primarily useful for diagnostic purposes.

Example

Consider the following prototype of a document from the `startup_log` collection:

```json
{
   "_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` 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` 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`.

---

9.4. Replication Reference 653
Collections on Replica Set Members

**local.system.replset**

`local.system.replset` (page 654) 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 654) 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 628) procedure. For additional information, see the `Oplog Size` (page 593) section.

**local.replset.minvalid**

This contains an object used internally by replica sets to track replication status.

**local.slaves**

*Removed in version 3.0:* Replica set members no longer mirror replication status of the set to the `local.slaves` (page 654) collection. Use `rs.status()` instead.

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`
    *Removed in version 3.0:* MongoDB no longer stores information about each slave in the `local.slaves` (page 654) collection. Use `db.serverStatus( { repl: 1 } )` instead.

- **On each slave:**
  - `local.sources`
    This contains information about the slave’s master server.

Replica Set Member States

Each member of a replica set has a state that reflects its disposition within the set.
<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>State Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STARTUP (page 655)</td>
<td>Not yet an active member of any set. All members start up in this state. The mongod parses the replica set configuration document (page 620) while in STARTUP (page 655). The member in state primary (page 564) is the only member that can accept write operations.</td>
</tr>
<tr>
<td>1</td>
<td>PRIMARY (page 655)</td>
<td>A member in state secondary (page 567) is replicating the data store. Data is available for reads, although they may be stale.</td>
</tr>
<tr>
<td>2</td>
<td>SECONDARY (page 655)</td>
<td>Can vote. Members either perform startup self-checks, or transition from completing a rollback (page 584) or resync (page 633).</td>
</tr>
<tr>
<td>3</td>
<td>RECOVERING (page 656)</td>
<td>The member has joined the set and is running an initial sync.</td>
</tr>
<tr>
<td>4</td>
<td>STARTUP2 (page 655)</td>
<td>The member’s state, as seen from another member of the set, is not yet known.</td>
</tr>
<tr>
<td>5</td>
<td>UNKNOWN (page 656)</td>
<td>Arbiter (page ??) do not replicate data and exist solely to participate in elections.</td>
</tr>
<tr>
<td>6</td>
<td>ARBITER (page 655)</td>
<td>The member, as seen from another member of the set, is unreachable.</td>
</tr>
<tr>
<td>7</td>
<td>DOWN (page 656)</td>
<td>This member is actively performing a rollback (page 584). Data is not available for reads.</td>
</tr>
<tr>
<td>8</td>
<td>ROLLBACK (page 656)</td>
<td>This member was once in a replica set but was subsequently removed.</td>
</tr>
<tr>
<td>9</td>
<td>REMOVED (page 656)</td>
<td>Each member of a replica set starts up in STARTUP (page 655) state. mongod then loads that member’s replica set configuration, and transitions the member’s state to STARTUP2 (page 655). Members in STARTUP (page 655) are not eligible to vote, as they are not yet a recognized member of any replica set.</td>
</tr>
<tr>
<td>10</td>
<td>REMOVE (page 656)</td>
<td>Each member of a replica set enters the STARTUP2 (page 655) 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</td>
</tr>
</tbody>
</table>

**States**

**Core States**

**PRIMARY**

Members in PRIMARY (page 655) state accept write operations. A replica set has at most one primary at a time. A SECONDARY (page 655) member becomes primary after an election (page 580). Members in the PRIMARY (page 655) state are eligible to vote.

**SECONDARY**

Members in SECONDARY (page 655) 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 655) state if the primary becomes unavailable.

**ARBITER**

Members in ARBITER (page 655) 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 655) 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 564) for more information on core states.

**Other States**

**STARTUP**

Each member of a replica set enters the STARTUP (page 655) 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 655) until all data is copied and all indexes are built. Afterwards, the member transitions to RECOVERING (page 656).

RECOVERING
A member of a replica set enters RECOVERING (page 656) state when it is not ready to accept reads. The RECOVERING (page 656) state can occur during normal operation, and doesn’t necessarily reflect an error condition. Members in the RECOVERING (page 656) state are eligible to vote in elections, but are not eligible to enter the PRIMARY (page 655) state.

A member transitions from RECOVERING (page 656) to SECONDARY (page 655) after replicating enough data to guarantee a consistent view of the data for client reads. The only difference between RECOVERING (page 656) and SECONDARY (page 655) states is that RECOVERING (page 656) prohibits client reads and SECONDARY (page 655) permits them. SECONDARY (page 655) 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 633) with the rest of the set. When this happens, the member enters the RECOVERING (page 656) state and requires manual intervention.

Error States   Members in any error state can’t vote.

UNKNOWN   Members that have never communicated status information to the replica set are in the UNKNOWN (page 656) state.

DOWN   Members that lose their connection to the replica set are seen as DOWN (page 656) by the remaining members of the set.

REMOVED   Members that are removed from the replica set enter the REMOVED (page 656) state. When members enter the REMOVED (page 656) 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 584), the member will have ROLLBACK (page 656) state.

FATAL   A member in FATAL (page 656) encountered an unrecoverable error. The member must be shut down and restarted; a resync may be required as well.

Read Preference Reference

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.

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.

25 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 130). The node that can complete \{w: majority\} (page 130) 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 657).
### Read Preference Modes

#### primary

All read operations use only the current replica set `primary`. This is the default read mode. If the primary is unavailable, read operations produce an error or throw an exception.

The `primary` (page 657) read preference mode is not compatible with read preference modes that use `tag sets` (page 590). If you specify a tag set with `primary` (page 657), the driver will produce an error.

#### primaryPreferred

In most situations, operations read from the `primary` but if it is unavailable, operations read from `secondary` members. All operations read from the `secondary` members of the replica set.

#### secondary

All operations read from the `secondary` members of the replica set.

#### secondaryPreferred

In most situations, operations read from `secondary` members but if no `secondary` members are available, operations read from the `primary`. Operations read from member of the `replica set` with the least network latency, irrespective of the member’s type.

#### nearest

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 130), 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 584).
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 590), 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 657) 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 657) 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 590), 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 591). If no secondaries have matching tags, the read operation produces an error.

Read operations using the secondary (page 658) 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 590), 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 591). If no secondaries have matching tags, the client ignores tags and reads from the primary.

Read operations using the secondaryPreferred (page 658) mode may return stale data.

**nearest**

The driver reads from the nearest member of the set according to the member selection (page 591) process. Reads in the nearest (page 658) mode do not consider the member's type. Reads in nearest (page 658) 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 590), 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 591).

Read operations using the nearest (page 658) 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 658) mode prefers low latency reads over a member’s primary or secondary status.

---

26 If your set has more than one secondary, and you use the secondary (page 658) read preference mode, consider the following effect. If you have a three member replica set (page 574) with a primary and two secondaries, and one secondary becomes unavailable, all secondary (page 658) queries must target the remaining secondary. This will double the load on this secondary. Plan and provide capacity to support this as needed.
For nearest (page 658), 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 591) 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 657) 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 ($p_{old}$) 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 ($p_{new}$), but for a brief period, the old primary ($p_{old}$) 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 ($p_{old}$) is still visible to clients as a primary, reads from this primary may reflect stale data.

- A primary ($p_{old}$) may become unresponsive, which will trigger an election and a new primary ($p_{new}$) can be elected, serving reads and writes. If the unresponsive primary ($p_{old}$) starts responding again, two primaries will 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 657). 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 660).

Minimize Latency To always read from a low-latency node, use nearest (page 658). The driver or mongos will read from the nearest member and those no more than 15 milliseconds $^{27}$ further away than the nearest member. nearest (page 658) 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 658) 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 635) `{‘dc’: ‘east’} and `{‘dc’: ‘west’}`, your application servers in the east data center can read from nearby members with the following read preference:

```javascript
db.collection.find().readPref({
    mode: 'nearest',
    tags: [{
        ‘dc’: ‘east’}
    ]}
)
```

$^{27}$ This threshold is configurable. See localPingThresholdMs for mongos or your driver documentation for the appropriate setting.

9.4. Replication Reference 659
Although nearest (page 658) 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 658). Although secondaryPreferred (page 658) 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 658) to distribute read load to replica sets, not secondaryPreferred (page 658).

**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 657) for the following commands:

- group
- mapReduce 28
- aggregate 29
- collStats
- dbStats
- 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.

---

28 Only “inline” mapReduce operations that do not write data support read preference, otherwise these operations must run on the primary members.

29 Using the $out pipeline operator forces the aggregation pipeline to run on the primary.
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 661) A high-level introduction to horizontal scaling, data partitioning, and sharded clusters in MongoDB.

**Sharding Concepts** (page 667) The core documentation of sharded cluster features, configuration, architecture and behavior.

  - **Sharded Cluster Components** (page 668) A sharded cluster consists of shards, config servers, and mongos instances.
  - **Sharded Cluster Architectures** (page 671) Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.
  - **Sharded Cluster Behavior** (page 674) 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 684) Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

**Sharded Cluster Tutorials** (page 690) Tutorials that describe common procedures and administrative operations relevant to the use and maintenance of sharded clusters.

**Sharding Reference** (page 736) Reference for sharding-related functions and operations.

### 10.1 Sharding Introduction

Sharding is a method for storing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

#### 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 563).

**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**.

**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

---

\(^1\) 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.
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 674) 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 682) 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 688), 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 684) 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)

---

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 661).

**Sharded Cluster Components** (page 668) A sharded cluster consists of shards, config servers, and mongos instances.

- **Shards** (page 669) A shard is a mongod instance that holds a part of the sharded collection’s data.
- **Config Servers** (page 670) Config servers hold the metadata about the cluster, such as the shard location of the data.

**Sharded Cluster Architectures** (page 671) Outlines the requirements for sharded clusters, and provides examples of several possible architectures for sharded clusters.

- **Sharded Cluster Requirements** (page 671) Discusses the requirements for sharded clusters in MongoDB.
- **Production Cluster Architecture** (page 672) Outlines the components required to deploy a redundant and highly available sharded cluster.

Continue reading from Sharded Cluster Architectures (page 671) for additional descriptions of sharded cluster deployments.

**Sharded Cluster Behavior** (page 674) 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 674) MongoDB uses the shard key to divide a collection’s data across the cluster’s shards.
- **Sharded Cluster High Availability** (page 676) Sharded clusters provide ways to address some availability concerns.
- **Sharded Cluster Query Routing** (page 678) The cluster’s routers, or mongos instances, send reads and writes to the relevant shard or shards.

**Sharding Mechanics** (page 684) Discusses the internal operation and behavior of sharded clusters, including chunk migration, balancing, and the cluster metadata.

- **Sharded Collection Balancing** (page 684) Balancing distributes a sharded collection’s data cluster to all of the shards.
- **Sharded Cluster Metadata** (page 689) The cluster maintains internal metadata that reflects the location of data within the cluster.

Continue reading from Sharding Mechanics (page 684) 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:

**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 669).

**Config Servers** Each config server (page 670) is a mongod instance that holds metadata about the cluster. The metadata maps chunks to shards. For more information, see [Config Servers](page 670).

**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 678).

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 691).
Shards

A shard is a replica set or a single mongod that contains a subset of the data for the sharded cluster. Together, the cluster’s shards hold the entire data set for the cluster.

Typically each shard is a replica set. The replica set provides redundancy and high availability for the data in each shard.

Important:  MongoDB shards data on a per collection basis. You must access all data in a sharded cluster via the mongos instances. If you connect directly to a shard, you will see only its fraction of the cluster's data. There is no 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” 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.

---

9 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

Config servers are special `mongodb` instances that store the metadata (page 689) 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 737). The `mongodb` 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 685), or
- after a chunk split (page 687).

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 `mongodb` will not perform auto-splits of chunks.

MongoDB reads data from the config server in the following cases:

- A new `mongodb` starts for the first time, or an existing `mongodb` 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 677) for more information.

### 10.2.2 Sharded Cluster Architectures

The following documents introduce deployment patterns for sharded clusters.

*Sharded Cluster Requirements* (page 671) Discusses the requirements for sharded clusters in MongoDB.

*Production Cluster Architecture* (page 672) Outlines the components required to deploy a redundant and highly available sharded cluster.

*Sharded Cluster Test Architecture* (page 672) Sharded clusters for testing and development can include fewer components.

### Sharded Cluster Requirements

While sharding is a powerful and compelling feature, sharded clusters have significant infrastructure requirements and increases the overall complexity of a deployment. As a result, only deploy sharded clusters when indicated by application and operational requirements.

Sharding is the *only* solution for some classes of deployments. Use *sharded clusters* if:

- your data set approaches or exceeds the storage capacity of a single MongoDB instance.
- the size of your system’s active working set will soon exceed the capacity of your system’s maximum RAM.
- a single MongoDB instance cannot meet the demands of your write operations, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add complexity to your system without adding much benefit.

**Important:** It takes time and resources to deploy sharding. If your system has already reached or exceeded its capacity, it will be difficult to deploy sharding without impacting your application.

As a result, if you think you will need to partition your database in the future, **do not** wait until your system is 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 684) will not begin moving data across shards until the imbalance of chunks among the shards exceeds the migration threshold (page 685). 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 688) 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 670) must be on separate machines. A single sharded cluster must have exclusive use of its config servers (page 670). 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 559).

- **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 691)

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 670).

- **At least one shard. Shards are either replica sets or a standalone mongod instances.**

- **One mongos instance.**

See

Production Cluster Architecture (page 672)
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 674)  MongoDB uses the shard key to divide a collection’s data across the cluster’s shards.

Sharded Cluster High Availability (page 676)  Sharded clusters provide ways to address some availability concerns.

Sharded Cluster Query Routing (page 678)  The cluster’s routers, or mongos instances, send reads and writes to the relevant shard or shards.

Tag Aware Sharding (page 682)  Tags associate specific ranges of shard key values with specific shards for use in managing deployment patterns.

Shard Keys

The shard key determines the distribution of the collection’s documents among the cluster’s shards. The shard key is either an indexed field or an indexed compound field that exists in every document in the collection.

MongoDB partitions data in the collection using ranges of shard key values. Each range, or chunk, defines a non-overlapping range of shard key values. MongoDB distributes the chunks, and their documents, among the shards in the cluster.

When a chunk grows beyond the chunk size (page 688), 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 521) 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 696).

**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 696), 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 676), 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 696).

**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 670), 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 678) and *config server* (page 670) 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 670). 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, 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 678) 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 678) and `Use Indexes to Sort Query Results` (page 548) 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 695) for more information.

Additional Information

- `Considerations for Selecting Shard Keys` (page 694)
- `Shard a Collection Using a Hashed Shard Key` (page 696).

Sharded Cluster High Availability

A production (page 672) 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.

---

10 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 486) page for more information on indexes and compound indexes.
**A Single mongod Becomes Unavailable in a Shard**

Replica sets (page 559) provide high availability for shards. If the unavailable mongod is a primary, then the replica set will elect (page 580) 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. \(^{11}\)

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 670).

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 684) or chunk splits (page 723) 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 684) nor will the mongos perform auto-chunk splits (page 723).

**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 670).

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

\(^{11}\) If an unavailable secondary becomes available while it still has current oplog entries, it can catch up to the latest state of the set using the normal replication process, otherwise it must perform an initial sync.
• to ensure that mongos can isolate most queries to a specific mongod.

Furthermore:
• Each shard should be a replica set, if a specific mongod instance fails, the replica set members will elect another to be primary and continue operation. However, if an entire shard is unreachable or fails for some reason, that data will be unavailable.
• If the shard key allows the mongos to isolate most operations to a single shard, then the failure of a single shard will only render some data unavailable.
• If your shard key distributes data required for every operation throughout the cluster, then the failure of the entire shard will render the entire cluster unavailable.

In essence, this concern for reliability simply underscores the importance of choosing a shard key that isolates query operations to a single shard.

Sharded Cluster Query Routing

MongoDB mongos instances route queries and write operations to shards in a sharded cluster. mongos provide the only interface to a sharded cluster from the perspective of applications. Applications never connect or communicate directly with the shards.

The mongos tracks what data is on which shard by caching the metadata from the config servers (page 670). 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:

```json
{ 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:
How `mongos` Handles Query Modifiers  If the result of the query is not sorted, the `mongos` instance opens a result cursor that “round robin” 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 }
```

![Diagram of MongoDB sharding concepts](image)

Depending on the distribution of data in the cluster and the selectivity of the query, `mongos` may still have to contact
multiple shards\textsuperscript{12} to fulfill these queries.

**Sharded and Non-Sharded Data**

Sharding operates on the collection level. You can shard multiple collections within a database or have multiple databases with sharding enabled.\textsuperscript{13} 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.

**Tag Aware Sharding**

MongoDB supports tagging a range of *shard key* values to associate that range with a shard or group of shards. Those shards receive all inserts within the tagged range.

The balancer obeys tagged range associations, which enables the following deployment patterns:

- isolate a specific subset of data on a specific set of shards.
- ensure that the most relevant data reside on shards that are geographically closest to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

\textsuperscript{12} *mongos* will route some queries, even some that include the shard key, to all shards, if needed.

\textsuperscript{13} 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.
Considerations

- Shard key range tags are distinct from replica set member tags (page 590).
- Hash-based sharding only supports tag-aware sharding on an 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 collection 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 those chunks to shards associated with those tags.

After configuring a tag 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 684).

See also:

*Manage Shard Tags* (page 730)

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments
- Webinar: Multi-Data Center Deployment

10.2.4 Sharding Mechanics

The following documents describe sharded cluster processes.

Sharded Collection Balancing (page 684) Balancing distributes a sharded collection’s data cluster to all of the shards.

Chunk Migration Across Shards (page 685) MongoDB migrates chunks to shards as part of the balancing process.

Chunk Splits in a Sharded Cluster (page 687) When a chunk grows beyond the configured size, MongoDB splits the chunk in half.

Shard Key Indexes (page 688) Sharded collections must keep an index that starts with the shard key.

Sharded Cluster Metadata (page 689) The cluster maintains internal metadata that reflects the location of data within the cluster.

Sharded Collection Balancing

Balancing is the process MongoDB uses to distribute data of a sharded collection evenly across a sharded cluster. When a shard has too many of a sharded collection’s chunks compared to other shards, MongoDB automatically balances the 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 737).

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 685) 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 686).

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 687).

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 686).
- 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 685).

You may disable the balancer temporarily for maintenance. See Disable the Balancer (page 717) 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 716) 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 715).

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.

<table>
<thead>
<tr>
<th>Number of Chunks</th>
<th>Migration Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer than 20</td>
<td>2</td>
</tr>
<tr>
<td>20-79</td>
<td>4</td>
</tr>
<tr>
<td>80 and greater</td>
<td>8</td>
</tr>
</tbody>
</table>

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 713) and Monitoring for MongoDB (page 187).

Chunk Migration Across Shards

Chunk migration moves the chunks of a sharded collection from one shard to another and is part of the balancer (page 684) 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 724) for more details.

- Automatic. The balancer (page 684) process automatically migrates chunks when there is an uneven distribution of a sharded collection’s chunks across the shards. See Migration Thresholds (page 685) 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 687).

   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 686).

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 685) 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.
The _waitForDelete, available as a setting for the balancer as well as the moveChunk command, can alter the behavior so that the delete phase of the current migration blocks the start of the next chunk migration. The _waitForDelete is generally for internal testing purposes. For more information, see Wait for Delete (page 714).

Chunk Migration and Replication  Changed in version 3.0: The default value secondaryThrottle became true for all chunk migrations.

New in version 3.0: The new writeConcern field in the balancer configuration document allows you to specify a write concern (page 78) semantics the _secondaryThrottle option.

By default, each document operation during chunk migration propagates to at least one secondary before the balancer proceeds with the next document, which is equivalent to a write concern of `{ w: 2 }`. You can set the writeConcern option on the balancer configuration to set different write concern semantics.

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 (page 714) to update the _secondaryThrottle parameter for the balancer.

For the moveChunk command, the secondaryThrottle parameter is independent of the _secondaryThrottle parameter for the balancer.

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 Manual16.

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.

Jumbo Chunks

During chunk migration, if the chunk exceeds the specified chunk size (page 688) 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 687) 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

As chunks grow beyond the specified chunk size (page 688) 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 684) for more details on balancing chunks across shards.

16http://docs.mongodb.org/v2.2/tutorial/configure-sharded-cluster-balancer/#sharded-cluster-config-secondary-throttle
Chunk Size

The default chunk size in MongoDB is 64 megabytes. You can *increase or reduce the chunk size* (page 727), 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 688) but cannot undergo a split; e.g. if a chunk represents a single shard key value. See *Considerations for Selecting Shard Keys* (page 694) 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.createIndex( { 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 670) 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 737).

**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 738)
- chunks (page 739)
- collections (page 740)
- databases (page 740)
- lockpings (page 740)
- locks (page 740)
- mongos (page 741)
- settings (page 741)
- shards (page 742)
- version (page 742)

For more information on these collections and their role in sharded clusters, see *Config Database* (page 737). See *Read and Write Operations on Config Servers* (page 670) 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 661).

**Sharded Cluster Deployment Tutorials** *(page 690)*  Instructions for deploying sharded clusters, adding shards, selecting shard keys, and the initial configuration of sharded clusters.

- **Deploy a Sharded Cluster** *(page 691)*  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 694)*  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 696)*  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 697)*  Add a shard to add capacity to a sharded cluster.

Continue reading from *Sharded Cluster Deployment Tutorials* *(page 690)* for additional tutorials.

**Sharded Cluster Maintenance Tutorials** *(page 705)*  Procedures and tasks for common operations on active sharded clusters.

- **View Cluster Configuration** *(page 706)*  View status information about the cluster’s databases, shards, and chunks.

- **Remove Shards from an Existing Sharded Cluster** *(page 719)*  Migrate a single shard’s data and remove the shard.

- **Migrate Config Servers with Different Hostnames** *(page 708)*  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 707)* procedure.

- **Manage Shard Tags** *(page 730)*  Use tags to associate specific ranges of shard key values with specific shards.

Continue reading from *Sharded Cluster Maintenance Tutorials* *(page 705)* for additional tutorials.

**Sharded Cluster Data Management** *(page 722)*  Practices that address common issues in managing large sharded data sets.

**Troubleshoot Sharded Clusters** *(page 735)*  Presents solutions to common issues and concerns relevant to the administration and use of sharded clusters. Refer to *FAQ: MongoDB Diagnostics* *(page 779)* 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 691)*  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 694)*  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 696)*  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 697)*  Add a shard to add capacity to a sharded cluster.
Deploy Three Config Servers for Production Deployments (page 698) 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 699) Convert a replica set to a sharded cluster in which each shard is its own replica set.

Convert Sharded Cluster to Replica Set (page 704) Replace your sharded cluster with a single replica set.

See also:
Enable Authentication in a Sharded Cluster (page 358)

Deploy a Sharded Cluster

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 735) 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 603) 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:27017" )
  ```

- To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

  ```
  sh.addShard( "mongodb0.example.net:27017" )
  ```

**Note:** It might take some time for `chunks` to migrate to the new shard.

### Enable Sharding for a Database

Before you can shard a collection, you must enable sharding for the collection’s database. Enabling sharding for a database does not redistribute data but make it possible to shard the collections in that database.

Once you enable sharding for a database, MongoDB assigns a primary shard for that database where MongoDB stores all data before sharding begins.

1. From a mongo shell, connect to the mongos instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

2. Issue the `sh.enableSharding()` method, specifying the name of the database for which to enable sharding. Use the following syntax:

   ```
   sh.enableSharding("<database>")
   ```

   Optionally, you can enable sharding for a database using the `enableSharding` command, which uses the following syntax:

   ```
   db.runCommand( { enableSharding: <database> } )
   ```

### Enable Sharding for a Collection

You enable sharding on a per-collection basis.

1. Determine what you will use for the shard key. Your selection of the shard key affects the efficiency of sharding. See the selection considerations listed in the Considerations for Selecting Shard Key (page 695).

2. If the collection already contains data you must create an index on the shard key using `createIndex()`. If the collection is empty then MongoDB will create the index as part of the `sh.shardCollection()` step.
3. Enable sharding for a collection by issuing the `sh.shardCollection()` method in the `mongo` shell. The method uses the following syntax:

```python
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:

```python
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 696) 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 696) 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 696) 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 521), 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 521) 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 687) occur.

See also:
Cardinality (page 696)

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 675)

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 676)

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 unequally 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 676) or appropriate write scaling (page 675).

If you choose a shard key with low cardinality, some chunks may grow too large for MongoDB to migrate. See Jumbo Chunks (page 687) 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

New in version 2.4.

Hashed shard keys (page 674) use a hashed index (page 521) 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 674). For limitations on hashed indexes, see Create a Hashed Index (page 521).
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
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

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 691).

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 684) 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 687).

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 mongod0.example.net, issue the following command:

     sh.addShard( "rs1/mongodb0.example.net:27017" )

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net:27017" )

   • To add a shard for a standalone mongod on port 27017 of mongodb0.example.net, issue the following command:

     sh.addShard( "mongodb0.example.net:27017" )

   Note: It might take some time for chunks to migrate to the new shard.

Deploy Three Config Servers for Production Deployments

This procedure converts a test deployment with only one config server (page 670) 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 661) 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 737), mongo-config0.example.net may resemble the following:

   rsync -az /data/configdb mongo-config1.example.net:/data/configdb
   rsync -az /data/configdb mongo-config2.example.net:/data/configdb

3. Start all three config servers, using the same invocation that you used for the single config server.

   mongod --configsvr

4. Restart all shard mongod and mongos processes.

Convert a Replica Set to a Replicated Sharded Cluster

Overview

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 700).
2. Start the config databases and a mongos. See Deploy Config Databases and mongos (page 700).
3. Add the initial replica set as a shard. See Add Initial Replica Set as a Shard (page 701).
4. Create a second shard and add to the cluster. See Add Second Shard (page 701).
5. Shard the desired collection. See Shard a Collection (page 702).

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 670).

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")
r.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.

```
use test

var bulk = db.test_collection.initializeUnorderedBulkOp();
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 603).

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:27019
```

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.

```javascript
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:

```javascript
sh.enableSharding( "test" )
```

The operation returns the status of the operation:

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

Step 3: Determine the shard key.  For the collection to shard, determine the shard key.  The `shard key` determines how MongoDB distributes the documents between shards.  Good shard keys:

- have values that are evenly distributed among all documents,
- group documents that are often accessed at the same time into contiguous chunks, and
- allow for effective distribution of activity among shards.

Once you shard a collection with the specified shard key, you **cannot** change the shard key.  For more information on shard keys, see `Shard Keys` and `Considerations for Selecting Shard Keys`.

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`.

```javascript
use test
db.test_collection.createIndex( { number : 1 } )
```
Step 5: Shard the collection. In the test database, shard the test_collection, specifying number as the shard key.

```javascript
use test
sh.shardCollection( "test.test_collection", { "number" : 1 } )
```

The method returns the status of the operation:

```javascript
{ "collectionssharded" : "test.test_collection", "ok" : 1 }
```

The balancer (page 684) 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.

```javascript
use test
db.stats()
db.printShardingStatus()
```

Example output of the db.stats():

```javascript
{
   "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" : 67108864,
         "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" : {
```
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.example.net:27017" }
  { "_id" : "rs1", "host" : "rs1/mongodb3.example.net:27017,mongodb4.example.net:27017,mongodb5.example.net:27017" }

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**

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 699). See the *Sharding* (page 661) 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 603) 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 249) 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 706) View status information about the cluster’s databases, shards, and chunks.

Migrate Config Servers with the Same Hostname (page 707) 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 708) 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 707) procedure.

Replace Disabled Config Server (page 709) 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 709) Migrate a sharded cluster to a different hardware system, for example, when moving a pre-production environment to production.

Backup Cluster Metadata (page 712) Create a backup of a sharded cluster’s metadata while keeping the cluster operational.

Configure Behavior of Balancer Process in Sharded Clusters (page 713) Manage the balancer’s behavior by scheduling a balancing window, changing size settings, or requiring replication before migration.

Manage Sharded Cluster Balancer (page 715) View balancer status and manage balancer behavior.

Remove Shards from an Existing Sharded Cluster (page 719) Migrate a single shard’s data and remove the shard.

View Cluster Configuration

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the databases collection in the Config Database (page 737). 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 ---

```json
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" : { "$minKey" : 1 } } -->> { "zip" : "56000" } on : shard0001 { "t" : 2, "i" : 0 }
  { "zip" : 56000 } -->> { "zip" : "56800" } on : shard0002 { "t" : 3, "i" : 4 }
  { "zip" : 56800 } -->> { "zip" : "57088" } on : shard0002 { "t" : 4, "i" : 2 }
  { "zip" : 57088 } -->> { "zip" : "57500" } on : shard0002 { "t" : 5, "i" : 3 }
  { "zip" : 57500 } -->> { "zip" : "58140" } on : shard0001 { "t" : 4, "i" : 0 }
  { "zip" : 58140 } -->> { "zip" : "59000" } on : shard0000 { "t" : 5, "i" : 1 }
  { "zip" : 59000 } -->> { "zip" : { "$maxKey" : 1 } } on : shard0000 { "t" : 3, "i" : 3 }

```

Migrate Config Servers with the Same Hostname

This procedure migrates a config server (page 670) in a sharded cluster (page 667) 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

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 670) in a sharded cluster (page 667) 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 707).

Considerations

Changing a config server’s (page 670) hostname requires downtime and requires restarting every process in the sharded cluster.

While migrating config servers, always make sure that all mongo 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 mongo instance’s configDB setting.

Procedure

1. Disable the cluster balancer process temporarily. See Disable the Balancer (page 717) 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:

   \[ \text{rsync -az /data/configdb mongodb.config2.example.net:/data/configdb} \]

4. Start the config server instance on the new system. The default invocation is:

   \[ \text{mongod --configsvr} \]

5. Shut down all existing MongoDB processes. This includes:
   
   - the mongod instances for the shards.
   - the mongod instances for the existing config databases (page 737).
   - 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 mongo instance.
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 717) section for more information on managing the balancer process.
Replace Disabled Config Server

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 670) in a sharded cluster (page 667). 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 708).

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

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 689). 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 614) 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 685) 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:

```bash
sh.stopBalancer()
```

To check the balancer state, issue the `sh.getBalancerState()` method.

For more information, see `Disable the Balancer` (page 717).

Migrate Each Config Server Separately

Migrate each config server (page 670) 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 708) 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:
   ```bash
   rsync -az /data/configdb/ mongodb.config2.example.net:/data/configdb
   ```

4. Start the config server instance on the new system. The default invocation is:
   ```bash
   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 692).

### 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 571) 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 652) 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 580) process. To step down the primary, connect to the primary and issue the either the replSetStepDown command or the rs.stepDown() method. The following example shows the rs.stepDown() method:
rs.stepDown()

2. Once the primary has stepped down and another member has become PRIMARY state. To migrate the stepped-down primary, follow the Migrate a Member of a Replica Set Shard (page 711) 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 614) and then use the procedure for migrating a replica set shard (page 711). 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 719) procedure.

**Re-Enable the Balancer**

To complete the migration, re-enable the balancer to resume chunk migrations (page 685).

Connect to one of the cluster’s mongos instances and pass true to the sh.setBalancerState() method:

```java
sh.setBalancerState(true)
```

To check the balancer state, issue the sh.getBalancerState() method.

For more information, see Enable the Balancer (page 718).

**Backup Cluster Metadata**

This procedure shuts down the mongod instance of a config server (page 670) in order to create a backup of a sharded cluster’s (page 661) 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 17.

1. Disable the cluster balancer process temporarily. See Disable the Balancer (page 717) 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 717) section for more information on managing the balancer process.

See also:

*MongoDB Backup Methods* (page 184).

17 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 670) for more information.
Configure Behavior of Balancer Process in Sharded Clusters

The balancer is a process that runs on one of the mongos instances in a cluster and ensures that chunks are evenly distributed throughout a sharded cluster. In most deployments, the default balancer configuration is sufficient for normal operation. However, administrators might need to modify balancer behavior depending on application or operational requirements. If you encounter a situation where you need to modify the behavior of the balancer, use the procedures described in this document.

For conceptual information about the balancer, see Sharded Collection Balancing (page 684) and Cluster Balancer (page 684).

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 716)
- Remove a Balancing Window Schedule (page 717).

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 688).

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 727).

Change the Maximum Storage Size for a Given Shard

The maxSize field in the shards (page 742) collection in the config database (page 737) sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If mapped size\(^{18}\) 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:

```javascript
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 742) collection in the config database (page 737). Set the maxSize in megabytes.

---

\(^{18}\) This value includes the mapped size of all data files including the ‘local’ and admin databases. Account for this when setting maxSize.
Example

Assume you have the following shard without a `maxSize` field:

```json
{ "_id" : "shard0000", "host" : "example.net:34001" }
```

Run the following sequence of commands in the `mongo` shell to insert a `maxSize` of 125 megabytes:

```javascript
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 125 } } )
```

To later increase the `maxSize` setting to 250 megabytes, run the following:

```javascript
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 250 } } )
```

Change Replication Behavior for Chunk Migration

Secondary Throttle  Changed in version 3.0.0: The balancer configuration document added configurable `writeConcern` to control the semantics of the `_secondaryThrottle` option.

The `_secondaryThrottle` parameter of the balancer and the `moveChunk` command affects the replication behavior during chunk migration (page 687). By default, `_secondaryThrottle` is `true`, which means each document move during chunk migration propagates to at least one secondary before the balancer proceeds with the next document: this is equivalent to a write concern of `{ w: 2 }`.

You can also configure the `writeConcern` for the `_secondaryThrottle` operation, to configure how migrations will wait for replication to complete. For more information on the replication behavior during various steps of chunk migration, see Chunk Migration and Replication (page 687).

To change the balancer’s `_secondaryThrottle` and `writeConcern` values, connect to a `mongos` instance and directly update the `_secondaryThrottle` value in the `settings` collection of the `config` database (page 737). For example, from a `mongo` shell connected to a `mongos`, issue the following command:

```javascript
use config
db.settings.update( { "_id" : "balancer" }, { $set : { "_secondaryThrottle" : false , "writeConcern": { "w": "majority" } } }, { upsert : true } )
```

The effects of changing the `_secondaryThrottle` and `writeConcern` value may not be immediate. To ensure an immediate effect, stop and restart the balancer to enable the selected value of `_secondaryThrottle`. See Manage Sharded Cluster Balancer (page 715) for details.

Wait for Delete  The `_waitForDelete` setting of the balancer and the `moveChunk` command affects how the balancer migrates multiple chunks from a shard. By default, the balancer does not wait for the on-going migration’s delete phase to complete before starting the next chunk migration. To have the delete phase `block` the start of the next chunk migration, you can set the `_waitForDelete` to true.

For details on chunk migration, see Chunk Migration (page 685). For details on the chunk migration queuing behavior, see Chunk Migration Queuing (page 686).

The `_waitForDelete` is generally for internal testing purposes. To change the balancer’s `_waitForDelete` value:

1. Connect to a `mongos` instance.
2. Update the `_waitForDelete` value in the `settings` (page 741) collection of the `config database` (page 737). For example:

```javascript
use config
db.settings.update(
   { "_id" : "balancer" },
   { $set : { "_waitForDelete" : true } },
   { upsert : true }
)
```

Once set to `true`, to revert to the default behavior:

1. Connect to a `mongos` instance.

2. Update or unset the `_waitForDelete` field in the `settings` (page 741) collection of the `config database` (page 737):

```javascript
use config
db.settings.update(
   { "_id" : "balancer", "_waitForDelete": true },
   { $unset : { "_waitForDelete" : "" } }
)
```

**Manage Sharded Cluster Balancer**

This page describes common administrative procedures related to balancing. For an introduction to balancing, see "Sharded Collection Balancing" (page 684). For lower level information on balancing, see "Cluster Balancer" (page 684).

See also:

*Configure Behavior of Balancer Process in Sharded Clusters* (page 713)

**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:

```javascript
sh.getBalancerState()
```

New in version 3.0.0: You can also see if the balancer is enabled using `sh.status()`. The `currently-enabled` field indicates whether the balancer is enabled, while the `currently-running` field indicates if the balancer is currently running.

**Check the Balancer Lock**

To see if the balancer process is active in your `cluster`, do the following:

1. Connect to any `mongos` in the cluster using the `mongo` shell.

2. Issue the following command to switch to the `Config Database` (page 737):

   ```javascript
   use config
   ```

3. Use the following query to return the balancer lock:
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 737):

   ```
   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 716) and wish to remove the schedule so that the balancer is always running, issue the following sequence of operations:

```javascript
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:

   ```javascript
   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:

   ```javascript
   sh.getBalancerState()
   ```

   Optionally, to verify no migrations are in progress after disabling, issue the following operation in the mongo shell:

   ```javascript
   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:

```javascript
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 184), 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 716) 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 717) 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:

```javascript
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:

```javascript
db.getSiblingDB("config").collections.findOne({_id : "students.grades"}).noBalance;
```

This operation will return a null error, `true`, `false`, or no output:

- A null error indicates the collection namespace is incorrect.
- If the result is `true`, balancing is disabled.
- If the result is `false`, balancing is enabled currently but has been disabled in the past for the collection. Balancing of this collection will begin the next time the balancer runs.
- If the operation returns no output, balancing is enabled currently and has never been disabled in the past for this collection. Balancing of this collection will begin the next time the balancer runs.

New in version 3.0.0: You can also see if the balancer is enabled using `sh.status()`. The `currently-enabled` field indicates if the balancer is enabled.

Remove Shards from an Existing Sharded Cluster

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 717).
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:

```json
{    "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:

```json
{    "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 721).

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:

   ```json
   { "_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:

   ```javascript
   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:

   ```json
   { "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:

```javascript
use admin
db.runCommand({ removeShard: "mongodb0" })
```

A success message appears at completion:

```json
{
   "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 254)
### 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 722)*: Create chunks, or *pre-split* empty collection to ensure an even distribution of chunks during data ingestion.
- **Split Chunks in a Sharded Cluster** *(page 723)*: Manually create chunks in a sharded collection.
- **Migrate Chunks in a Sharded Cluster** *(page 724)*: Manually migrate chunks without using the automatic balance process.
- **Merge Chunks in a Sharded Cluster** *(page 725)*: Use the `mergeChunks` to manually combine chunk ranges.
- **Modify Chunk Size in a Sharded Cluster** *(page 727)*: Modify the default chunk size in a sharded collection.
- **Clear jumbo Flag** *(page 728)*: Clear *jumbo* flag from a shard.
- **Manage Shard Tags** *(page 730)*: Use tags to associate specific ranges of shard key values with specific shards.
- **Enforce Unique Keys for Sharded Collections** *(page 732)*: Ensure that a field is always unique in all collections in a sharded cluster.
- **Shard GridFS Data Store** *(page 734)*: 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 684) and Chunk Migration (page 685). For information on manually migrating chunks, see Migrate Chunks in a Sharded Cluster (page 724).

Split Chunks in a Sharded Cluster

Normally, MongoDB splits a chunk after an insert if the chunk exceeds the maximum chunk size (page 688). 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 725) 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:

```javascript
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.

```javascript
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 716), 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 684) and *Chunk Migration* (page 685).

**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.

```javascript
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:

```javascript
var shServer = [ "sh0.example.net", "sh1.example.net", "sh2.example.net", "sh3.example.net", "sh4.example.net" ];
for ( var x=97; x<97+26; x++ ){
   for( var y=97; y<97+26; y+=6 ) {
```
var prefix = String.fromCharCode(x) + String.fromCharCode(y);
    db.adminCommand({moveChunk : "myapp.users", find : {email : prefix}, to : shServer[(y-97)/6]})
}
}

See Create Chunks in a Sharded Cluster (page 722) for an introduction to pre-splitting.

New in version 2.2: The moveChunk command has the: `_secondaryThrottle` parameter. When set to `true`, MongoDB ensures that changes to shards as part of chunk migrations replicate to secondaries throughout the migration operation. For more information, see Change Replication Behavior for Chunk Migration (page 714).

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

**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 722) 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` field 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,
```

10.3. Sharded Cluster Tutorials
"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" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard0000
{ "username" : "user16643" } -->> { "username" : "user2329" } on : shard0000
{ "username" : "user2329" } -->> { "username" : "user29937" } on : shard0000
{ "username" : "user29937" } -->> { "username" : "user36583" } on : shard0000
{ "username" : "user36583" } -->> { "username" : "user43229" } on : shard0000
{ "username" : "user43229" } -->> { "username" : "user49877" } on : shard0000
{ "username" : "user49877" } -->> { "username" : "user56522" } on : shard0000
{ "username" : "user56522" } -->> { "username" : "user63169" } on : shard0000
{ "username" : "user63169" } -->> { "username" : "user69816" } on : shard0000
{ "username" : "user69816" } -->> { "username" : "user76462" } on : shard0000
{ "username" : "user76462" } -->> { "username" : "user83108" } on : shard0000
{ "username" : "user83108" } -->> { "username" : "user89756" } on : shard0000
{ "username" : "user89756" } -->> { "username" : "user96401" } on : shard0000
{ "username" : "user96401" } -->> { "username" : { "$maxKey" : 1 } } on : shard0000

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": "user6982" }, 
               { "username": "user96197" } ]
} )

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" : { "$minKey" : 1 } } -->> { "username" : "user16643" } on : shard0000 Timestamp(2, 0)
    { "username" : "user16643" } -->> { "username" : "user56522" } on : shard0000 Timestamp(3, 0)
    { "username" : "user56522" } -->> { "username" : "user96401" } on : shard0001 Timestamp(8, 1)
    { "username" : "user96401" } -->> { "$maxKey" : 1 } on : shard0000
```

**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 737):

   ```
   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

If MongoDB cannot split a chunk that exceeds the specified chunk size (page 688) or contains a number of documents that exceeds the max, MongoDB labels the chunk as jumbo (page 687).

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:
Step 3: Split the jumbo Chunk. Use either \texttt{sh.splitAt()} or \texttt{sh.splitFind()} to split the jumbo chunk.

\begin{verbatim}
sh.splitAt("test.foo", \{ x: 3 \})
\end{verbatim}

MongoDB removes the jumbo flag upon successful split of the chunk.

\textbf{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.

\textbf{Important:} Only use this method if the preferred method (page 728) is not applicable.

Before modifying the \textit{config database} (page 737), 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.

\textbf{Step 1: Stop the balancer.} Disable the cluster balancer process temporarily, following the steps outlined in \textit{Disable the Balancer} (page 717).

\textbf{Step 2: Create a backup of config database.} Use \texttt{mongodump} against a config server to create a backup of the \textit{config database}. For example:

\begin{verbatim}
mongodump --db config --port <config server port> --out <output file>
\end{verbatim}

\textbf{Step 3: Connect to mongos.} Connect a \texttt{mongo} shell to a \texttt{mongos}.

\textbf{Step 4: Find the jumbo Chunk.} Run \texttt{sh.status(true)} to find the chunk labeled jumbo.

\begin{verbatim}
sh.status(true)
\end{verbatim}

For example, the following output from \texttt{sh.status(true)} shows that chunk with shard key range \{ "x" : 2 \} \rightarrow \{ "x" : 3 \} is jumbo.

\begin{verbatim}
--- Sharding Status ---
sharding version: {
  ...
}
shards:
  ...
databases:
  ...
    test.foo
      shard key: { "x" : 1 }
      chunks:
        shard-b 2
\end{verbatim}
Step 5: Update chunks collection. In the chunks collection of the config database, unset the jumbo flag for the chunk. For example,

```javascript
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 718).

Step 7: Optional. Clear current cluster meta information. To ensure that mongos instances update their cluster information cache, run flushRouterConfig in the admin database.

```javascript
db.adminCommand({ flushRouterConfig: 1 } )
```

Manage Shard Tags

In a sharded cluster, you can use tags to associate specific ranges of a shard key with a specific shard or subset of shards.

Tag a Shard

Associate tags with a particular shard using the sh.addShardTag() method when connected to a mongos instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag NYC to two shards, and the tags SFO and NRT to a third shard:

```javascript
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:

```javascript
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range

To assign a tag to a range of shard keys use the sh.addTagRange() method when connected to a mongos instance. Any given shard key range may only have one assigned tag. You cannot overlap defined ranges, or tag the same range more than once.
Example

Given a collection named users in the records database, sharded by the zipcode field. The following operations assign:

- two ranges of zip codes in Manhattan and Brooklyn the NYC tag
- one range of zip codes in San Francisco the SFO tag

```javascript
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 742) collection of the config database.

Each document in the tags (page 742) 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:

```javascript
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 742) 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:

```javascript
use config
db.shards.find({ tags: "NYC" })
```

You can find tag ranges for all namespaces in the tags (page 742) 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:

```javascript
use config
db.tags.find({ tags: "NYC" })
```

Additional Resource

- Whitepaper: MongoDB Multi-Data Center Deployments\(^\text{19}\)
- Webinar: Multi-Data Center Deployment\(^\text{20}\)

\(^\text{19}\)https://www.mongodb.com/lp/white-paper/multi-dc?jmp=docs
Enforce Unique Keys for Sharded Collections

Overview

The unique constraint on indexes ensures that only one document can have a value for a field in a collection. For sharded collections these unique indexes cannot enforce uniqueness because insert and indexing operations are local to each shard.

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 674).
   
   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 502).

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:

\[
\text{db.runCommand( \{ shardCollection : "test.users" , key : \{ email : 1 \} , unique : true \} )};
\]

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:

\[
\text{db.runCommand( \{ shardCollection : "test.users" \} )}
\]

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 675) and query isolation (page 676), as well as high cardinality (page 696). 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 732); otherwise, you can simply create multiple unique indexes on the collection.

Process  Consider the following for the “proxy collection:”

```json
{
    "_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:

```javascript
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:

```javascript
db.proxy.createIndex( { "email" : 1 } , { unique : true } )
```

You may create multiple unique indexes on this collection if you do not plan to shard the proxy collection.

To insert documents, use the following procedure in the JavaScript shell:

```javascript
db = db.getSiblingDB('records');
var primary_id = ObjectId();

db.proxy.insert({
    "_id" : primary_id
    "email" : "example@example.net"
});

// if: the above operation returns successfully,
// then continue:

db.information.insert({
    "_id" : primary_id
    "email": "example@example.net"
    // additional information...
});
```

You must insert a document into the proxy collection first. If this operation succeeds, the email field is unique, and you may continue by inserting the actual document into the information collection.

See
The full documentation of: createIndex() and shardCollection.

Considerations
• Your application must catch errors when inserting documents into the “proxy” collection and must enforce consistency between the two collections.
• If the proxy collection requires sharding, you must shard on the single field on which you want to enforce uniqueness.
• To enforce uniqueness on more than one field using sharded proxy collections, you must have one proxy collection for every field for which to enforce uniqueness. If you create multiple unique indexes on a single proxy collection, you will not be able to shard proxy collections.

Use Guaranteed Unique Identifier The best way to ensure a field has unique values is to generate universally unique identifiers (UUID,) such as MongoDB’s ObjectId values.

This approach is particularly useful for the _id field, which must be unique: for collections where you are not sharding by the _id field the application is responsible for ensuring that the _id field is unique.

Shard GridFS Data Store

When sharding a GridFS store, consider the following:

files Collection

Most deployments will not need to shard the files collection. The files collection is typically small, and only contains metadata. None of the required keys for GridFS lend themselves to an even distribution in a sharded situation. If you must shard the files collection, use the _id field possibly in combination with an application field.

Leaving files unsharded means that all the file metadata documents live on one shard. For production GridFS stores you must store the files collection on a replica set.

chunks Collection

To shard the chunks collection by { files_id : 1 , n : 1 }, issue commands similar to the following:

db.fs.chunks.createIndex( { files_id : 1 , n : 1 } )

db.runCommand( { shardCollection : "test.fs.chunks" , key : { files_id : 1 , n : 1 } } )

You may also want to shard using just the file_id field, as in the following operation:

db.runCommand( { shardCollection : "test.fs.chunks" , key : { file_id : 1 } } )

Important: { files_id : 1 , n : 1 } and { files_id : 1 } are the only supported shard keys for the chunks collection of a GridFS store.

Note: Changed in version 2.2.

Before 2.2, you had to create an additional index on files_id to shard using only this field.

The default files_id value is an ObjectId, as a result the values of files_id are always ascending, and applications will insert all new GridFS data to a single chunk and shard. If your write load is too high for a single server to handle, consider a different shard key or use a different value for _id in the files collection.
10.3.4 Troubleshoot Sharded Clusters

This section describes common strategies for troubleshooting sharded cluster deployments.

Config Database String Error

Start all mongos instances in a sharded cluster with an identical configDB string. If a mongos instance tries to connect to the sharded cluster with a configDB string that does not exactly match the string used by the other mongos instances, including the order of the hosts, the following errors occur:

could not initialize sharding on connection

And:

mongos specified a different config database string

To solve the issue, restart the mongos with the correct string.

Cursor Fails Because of Stale Config Data

A query returns the following warning when one or more of the mongos instances has not yet updated its cache of the cluster’s metadata from the config database:

could not initialize cursor across all shards because : stale config detected

This warning should not propagate back to your application. The warning will repeat until all the mongos instances refresh their caches. To force an instance to refresh its cache, run the flushRouterConfig command.

Avoid Downtime when Moving Config Servers

Use CNAMEs to identify your config servers to the cluster so that you can rename and renumber your config servers without downtime.
10.4 Sharding Reference

10.4.1 Sharding Methods in the mongo Shell

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh._adminCommand()</td>
<td>Runs a database command against the admin database, like db.runCommand(), but can confirm that it is issued against a mongos.</td>
</tr>
<tr>
<td>sh.getBalancerLockDetails()</td>
<td>Reports on the active balancer lock, if it exists.</td>
</tr>
<tr>
<td>sh._checkFullName()</td>
<td>Tests a namespace to determine if its well formed.</td>
</tr>
<tr>
<td>sh._checkMongos()</td>
<td>Tests to see if the mongo shell is connected to a mongos instance.</td>
</tr>
<tr>
<td>sh._lastMigration()</td>
<td>Reports on the last chunk migration.</td>
</tr>
<tr>
<td>sh.addShard()</td>
<td>Adds a shard to a sharded cluster.</td>
</tr>
<tr>
<td>sh.addShardTag()</td>
<td>Associates a shard with a tag, to support tag aware sharding (page 682).</td>
</tr>
<tr>
<td>sh.addTagRange()</td>
<td>Associates range of shard keys with a shard tag, to support tag aware sharding (page 682).</td>
</tr>
<tr>
<td>sh.removeTagRange()</td>
<td>Removes an association between a range shard keys and a shard tag. Use to manage tag aware sharding (page 682).</td>
</tr>
<tr>
<td>sh.getBalancerHost()</td>
<td>Returns the name of a mongos that’s responsible for the balancer process.</td>
</tr>
<tr>
<td>sh.getBalancerState()</td>
<td>Returns a boolean to report if the balancer is currently enabled.</td>
</tr>
<tr>
<td>sh.help()</td>
<td>Returns help text for the sh methods.</td>
</tr>
<tr>
<td>sh.isBalancerRunning()</td>
<td>Returns a boolean to report if the balancer process is currently migrating chunks.</td>
</tr>
<tr>
<td>sh.moveChunk()</td>
<td>Migrates a chunk in a sharded cluster.</td>
</tr>
<tr>
<td>sh.removeShardTag()</td>
<td>Removes the association between a shard and a shard tag.</td>
</tr>
<tr>
<td>sh.setBalancerState()</td>
<td>Enables or disables the balancer which migrates chunks between shards.</td>
</tr>
<tr>
<td>sh.shardCollection()</td>
<td>Enables sharding for a collection.</td>
</tr>
<tr>
<td>sh.splitAt()</td>
<td>Divides an existing chunk into two chunks using a specific value of the shard key as the dividing point.</td>
</tr>
<tr>
<td>sh.splitFind()</td>
<td>Divides an existing chunk that contains a document matching a query into two approximately equal chunks.</td>
</tr>
<tr>
<td>sh.startBalancer()</td>
<td>Enables the balancer and waits for balancing to start.</td>
</tr>
<tr>
<td>sh.status()</td>
<td>Reports on the status of a sharded cluster, as db.printShardingStatus().</td>
</tr>
<tr>
<td>sh.stopBalancer()</td>
<td>Disables the balancer and waits for any in progress balancing rounds to complete.</td>
</tr>
<tr>
<td>sh.waitForBalancer()</td>
<td>Internal. Waits for the balancer state to change.</td>
</tr>
<tr>
<td>sh.waitForDLock()</td>
<td>Internal. Waits until the balancer stops running.</td>
</tr>
<tr>
<td>sh.waitForPingChange()</td>
<td>Internal. Waits for a specified distributed sharded cluster lock.</td>
</tr>
</tbody>
</table>

10.4.2 Sharding Database Commands

The following database commands support sharded clusters.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flushRouterConfig</td>
<td>Forces an update to the cluster metadata cached by a mongos.</td>
</tr>
<tr>
<td>addShard</td>
<td>Adds a shard to a sharded cluster.</td>
</tr>
<tr>
<td>cleanupOrphaned</td>
<td>Removes orphaned data with shard key values outside of the ranges of the chunks owned by a shard.</td>
</tr>
<tr>
<td>checkShardingIndex</td>
<td>Internal command that validates index on shard key.</td>
</tr>
<tr>
<td>enableSharding</td>
<td>Enables sharding on a specific database.</td>
</tr>
<tr>
<td>listShards</td>
<td>Returns a list of configured shards.</td>
</tr>
<tr>
<td>removeShard</td>
<td>Starts the process of removing a shard from a sharded cluster.</td>
</tr>
<tr>
<td>getShardMap</td>
<td>Internal command that reports on the state of a shard.</td>
</tr>
<tr>
<td>getShardVersion</td>
<td>Internal command that returns the config server version.</td>
</tr>
<tr>
<td>mergeChunks</td>
<td>Provides the ability to combine chunks on a single shard.</td>
</tr>
<tr>
<td>setShardVersion</td>
<td>Internal command to sets the config server version.</td>
</tr>
<tr>
<td>shardCollection</td>
<td>Enables the sharding functionality for a collection, allowing the collection to be sharded.</td>
</tr>
<tr>
<td>shardingState</td>
<td>Reports whether the mongod is a member of a sharded cluster.</td>
</tr>
<tr>
<td>unsetSharding</td>
<td>Internal command that affects connections between instances in a MongoDB deployment.</td>
</tr>
<tr>
<td>split</td>
<td>Creates a new chunk.</td>
</tr>
<tr>
<td>splitChunk</td>
<td>Internal command to split chunk. Instead use the methods sh.splitFind() and sh.splitAt().</td>
</tr>
<tr>
<td>splitVector</td>
<td>Internal command that determines split points.</td>
</tr>
<tr>
<td>medianKey</td>
<td>Deprecated internal command. See splitVector.</td>
</tr>
<tr>
<td>moveChunk</td>
<td>Internal command that migrates chunks between shards.</td>
</tr>
<tr>
<td>movePrimary</td>
<td>Reassigns the primary shard when removing a shard from a sharded cluster.</td>
</tr>
<tr>
<td>isdbgrid</td>
<td>Verifies that a process is a mongos.</td>
</tr>
</tbody>
</table>

### 10.4.3 Reference Documentation

**Config Database** *(page 737)*  
Complete documentation of the content of the local database that MongoDB uses to store sharded cluster metadata.

**Config Database**

The config database supports sharded cluster operation. See the [Sharding](page 661) 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:

```bash
use config
```

You can return a list of the collections, with the following helper:

```bash
show collections
```
Collections

**config**

`config.changelog`

---

**Internal MongoDB Metadata**

The **config** (page 738) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The **changelog** (page 738) 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 738) collection:

```json
{
   "_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("000000000000000000000000000000000"
      },
      "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 738) 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 739) document depends on the type of change.

**config.chunks**

**Internal MongoDB Metadata**

The config (page 738) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The chunks (page 739) 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\":

```json
{
    "_id" : "mydb.foo-a\"cat\"",
    "lastmod" : Timestamp(1000, 3),
    "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fde"),
    "ns" : "mydb.foo",
    "min" : {
        "animal" : "cat"
    },
    "max" : {
        "animal" : "dog"
    }
}
```
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.

Internal MongoDB Metadata

The `config` database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `collections` 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` collection would resemble the following:

```json
{
   "_id" : "records.pets",
   "lastmod" : ISODate("1970-01-16T15:00:58.107Z"),
   "dropped" : false,
   "key" : {
      "a" : 1
   },
   "unique" : false,
   "lastmodEpoch" : ObjectId("5078407bd58b175c5c225fde")
}
```

Internal MongoDB Metadata

The `databases` collection stores a document for each database in the cluster, and tracks if the database has sharding enabled. `databases` represents each database in a distinct document. When a databases have sharding enabled, the `primary` field holds the name of the primary shard.

```json
{ "_id" : "admin", "partitioned" : false, "primary" : "config" }
{ "_id" : "mydb", "partitioned" : true, "primary" : "shard0000" }
```

Internal MongoDB Metadata

The `lockpings` 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` collection would resemble:

```json
{ "_id" : "example.com:30000:1350047994:16807", "ping" : ISODate("2012-10-12T18:32:54.892Z") }
```
config.locks

**Internal MongoDB Metadata**

The `config` (page 738) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `locks` (page 740) 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.

```json
{
    "_id" : "balancer",
    "process" : "example.net:40000:1350402818:16807",
    "state" : 2,
    "ts" : ObjectId("507daeedf40e1879df62e5f3"),
    "when" : ISODate("2012-10-16T19:01:01.593Z"),
    "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 738) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `mongos` (page 741) 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`.

```json
{
    "_id" : "example.com:30000",
    "ping" : ISODate("2012-10-12T17:08:13.538Z"),
    "up" : 13699,
    "waiting" : true
}
```

config.settings

**Internal MongoDB Metadata**

The `config` (page 738) database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `settings` (page 741) collection holds the following sharding configuration settings:

- Chunk size. To change chunk size, see *Modify Chunk Size in a Sharded Cluster* (page 727).
- Balancer status. To change status, see *Disable the Balancer* (page 717).

The following is an example `settings` collection:
Internal MongoDB Metadata

The `config` database is internal: applications and administrators should not modify or depend upon its content in the course of normal operation.

The `shards` 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` 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" ] }
```

Internal MongoDB Metadata

The `tags` collection holds documents for each tagged shard key range in the cluster. The documents in the `tags` collection resemble the following:

```
{ 
   "_id" : { "ns" : "records.users", "min" : { "zipcode" : "10001" } },
   "ns" : "records.users",
   "min" : { "zipcode" : "10001" },
   "max" : { "zipcode" : "10281" },
   "tag" : "NYC"
}
```

Internal MongoDB Metadata

The `version` collection holds the current metadata version number. This collection contains only one document:

To access the `version` 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 }
```
CHAPTER 11

Frequently Asked Questions

11.1 FAQ: MongoDB Fundamentals

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 743) or post your question to the MongoDB User Mailing List.

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, atomic writes on a per-document level, and fully-consistent reads.

Operationally, MongoDB features master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

Note: MongoDB uses BSON, a binary object format similar to, but more expressive than JSON.

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 131)

1https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
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 131)

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 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.

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 758).

https://docs.mongodb.org/ecosystem/drivers
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 779) for answers to additional questions about MongoDB and Memory use.

11.1.9 How do I configure the cache size for MMAPv1?

MongoDB has no configurable cache for MMAPv1 storage engine. 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.

For the WiredTiger storage engine, you can specify the maximum size of the cache that WiredTiger will use for all data. See `storage.wiredTiger.engineConfig.cacheSizeGB` and `--wiredTigerCacheSizeGB`.

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 308) 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?

Changed in version 3.0: Commercial support is no longer provided for MongoDB on 32-bit platforms (Linux and Windows). See Platform Support (page 809).

32-bit versions of MongoDB do not support the WiredTiger storage engine.

When running a 32-bit build of MongoDB, the total storage size for the server, including data and indexes, is 2 gigabytes. For this reason, do not deploy MongoDB to production on 32-bit machines.

If you’re running a 64-bit build of MongoDB, there’s virtually no limit to storage size. For production deployments, 64-bit builds and operating systems are strongly recommended.

See also:

“Blog Post: 32-bit Limitations”

Note: 32-bit builds disable journaling by default because journaling further limits the maximum amount of data that the database can store.

11.2 FAQ: MongoDB for Application Developers

This document answers common questions about application development using MongoDB.

If you don’t find the answer you’re looking for, check the complete list of FAQs (page 743) or post your question to the MongoDB User Mailing List.

11.2.1 What is a namespace in MongoDB?

A “namespace” is the concatenation of the database name and the collection names with a period character in between.

Collections are containers for documents that share one or more indexes. Databases are groups of collections stored on disk using a single set of data files.

For an example acme.users namespace, acme is the database name and users is the collection name. Period characters can occur in collection names, so that 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.

Note:

Each index also has its own namespace.

MongoDB database have a configurable limit on the number of namespaces in a database.
11.2.2 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use \texttt{remove()}, the object will no longer exist in MongoDB’s on-disk data storage.

11.2.3 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 \textit{journal} roughly every 100 milliseconds. These values are configurable with the \texttt{commitIntervalMs} and \texttt{syncPeriodSecs}.

These values represent the \textit{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 \texttt{fsync} and \textit{Write Concern Reference} (page 130) to modify this strategy.

11.2.4 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 \textit{partitioned} or \textit{replicated} systems with a number of database server processes.

MongoDB \textit{does} have support for atomic operations \textit{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 \textit{Atomicity and Transactions} (page 80) page.

11.2.5 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new \textit{aggregation framework} (page 439), with the \texttt{aggregate} command.

MongoDB also supports \textit{map-reduce} with the \texttt{mapReduce} command, as well as basic aggregation with the \texttt{group}, \texttt{count}, and \texttt{distinct} commands.

See also:

The \textit{Aggregation} (page 435) page.

11.2.6 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 \texttt{quiet} option or the command-line option \texttt{--quiet} to suppress these messages from the log.
11.2.7 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

11.2.8 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 774)

11.2.9 How do I optimize storage use for small documents?

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the _id field explicitly.
  
  MongoDB clients automatically add an _id field to each document and generate a unique 12-byte ObjectId for the _id field. Furthermore, MongoDB always indexes the _id field. For smaller documents this may account for a significant amount of space.

  To optimize storage use, users can specify a value for the _id field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the _id field that would have occupied space in another portion of the document.

  You can store any value in the _id field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field’s value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.
  
  MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of documents that resemble the following:

  ```
  { last_name : "Smith", best_score: 3.9 }
  ```

  If you shorten the 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.

  7https://docs.mongodb.org/ecosystem/platforms/amazon-ec2
• Embed documents.

In some cases you may want to embed documents in other documents and save on the per-document overhead.

### 11.2.10 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 579) 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 150).

### 11.2.11 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:

```cpp
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`
- `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.

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” page in the PHP driver documentation for more information.

11.2.12 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 758).

11.2.13 What is the compare order for BSON types?

MongoDB permits documents within a single collection to have fields with different BSON types. For instance, the following documents may exist within a single collection.

```json
{ 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
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Changed in version 3.0.0: Date objects sort before Timestamp objects. Previously Date and Timestamp objects sorted together.

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.

Consider the following mongo example:

```javascript
db.test.insert( {x : 3 } );
db.test.insert( {x : 2.9 } );
db.test.insert( {x : new Date() } );
db.test.insert( {x : true } );
```
Examples:

```javascript
db.test.find().sort({x:1});
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154ce8de6586fb002c6"), "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 123) page for an example of a C++ use of *MinKey*.

### 11.2.14 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:

<table>
<thead>
<tr>
<th></th>
<th>32-bit Integer</th>
<th>64-bit Integer</th>
<th>Float</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit Integer</td>
<td>32-bit or 64-bit Integer</td>
<td>64-bit Integer</td>
<td>Float</td>
</tr>
<tr>
<td>64-bit Integer</td>
<td>64-bit Integer</td>
<td>64-bit Integer</td>
<td>Float</td>
</tr>
<tr>
<td>Float</td>
<td>Float</td>
<td>Float</td>
<td>Float</td>
</tr>
</tbody>
</table>

**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.

### 11.2.15 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:

```javascript
{ _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 505), 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 505) for more information.

Given the following query:
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.

### 11.2.16 Are there any restrictions on the names of Collections?

Collection names can be any UTF-8 string with the following exceptions:
- A collection name should begin with a letter or an underscore.
- The empty string ("") is not a valid collection name.
- Collection names cannot contain the $ character. (version 2.2 only)
- Collection names cannot contain the null character: \
- Do not name a collection using the `system` prefix. MongoDB reserves `system` for system collections.
- 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\(^9\).

---

\(^9\)https://api.mongodb.org/

---

11.2. FAQ: MongoDB for Application Developers 752
To create a collection `_foo` and insert the `{ a : 1 }` document, use the following operation:

```javascript
db.getCollection("_foo").insert( { a : 1 } )
```

To perform a query, use the `find()` method, in as the following:

```javascript
db.getCollection("_foo").find()
```

### 11.2.17 How do I isolate cursors from intervening write operations?

MongoDB cursors can return the same document more than once in some situations. 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.

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.

### 11.2.18 When should I embed documents within other documents?

When modeling data in MongoDB (page 145), embedding is frequently the choice for:

- “contains” relationships between entities.
- one-to-many relationships when the “many” objects always appear with or are viewed in the context of their parents.

You should also consider embedding for performance reasons if you have a collection with a large number of small documents. Nevertheless, if small, separate documents represent the natural model for the data, then you should maintain that model.

If, however, you can group these small documents by some logical relationship and you frequently retrieve the documents by this grouping, you might consider “rolling-up” the small documents into larger documents that contain an array of 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 486) for more information on indexes.

---

10 As a cursor returns documents other operations may interleave with the query: if some of these operations are `updates` (page 73) 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.
11.2.19 Where can I learn more about data modeling in MongoDB?

Begin by reading the documents in the *Data Models* (page 143) 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.

11.2.20 Can I manually pad documents to prevent moves during updates?

Changed in version 3.0.0.

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 by default, MongoDB uses *Power of 2 Sized Allocations* (page 91) to add *padding automatically* (page 91). The *Power of 2 Sized Allocations* (page 91) 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 as well as reduce the occurrences of reallocations in many cases.

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.

```javascript
var myTempPadding = [  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" ];

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 91)

---

\(^{12}\)http://www.mongodb.com/presentations/mongodb-melbourne-2012/schema-design-example
^{13}\)http://dmer.rumblr.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
11.3 FAQ: The mongo Shell

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:

  ```javascript
  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:

  ```javascript
  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:

   ```bash
   EDITOR=vim
   ```

2. Start the mongo shell:

   ```bash
   mongo
   ```

3. Define a function myFunction:

   ```javascript
   function myFunction () {} 
   ```

4. Edit the function using your editor:

   ```bash
   edit myFunction
   ```

   The command should open the vim edit session. Remember to save your changes.

5. Type myFunction to see the function definition:

   ```bash
   myFunction
   ```

   The result should be the changes from your saved edit:

   ```javascript
   function myFunction() {
       print("This was edited");
   }
   ```
11.4 FAQ: Concurrency

Changed in version 3.0.

MongoDB allows multiple clients to read and write the same data. In order to ensure consistency, it uses locking and other concurrency control measures to prevent multiple clients from modifying the same piece of data simultaneously. Together, these mechanisms guarantee that all writes to a single document occur either in full or not at all and that clients never see an inconsistent view of the data.

11.4.1 What type of locking does MongoDB use?

MongoDB uses reader-writer locks that allow concurrent readers shared access to a resource, such as a database or collection, but give exclusive access to a single write operation.

MongoDB uses multi-granularity locking that allows operations to lock at the global, database or collection level, and allows for individual storage engines to implement their own concurrency control below the collection (i.e., at the document-level in WiredTiger).

In addition to a shared (S) locking mode for reads and an exclusive (X) locking mode for write operations, intent shared (IS) and intent exclusive (IX) modes indicate an intent to read or write a resource using a finer granularity lock. When locking at a certain granularity all higher levels are locked using an intent lock.

For example, when locking a collection for writing (using mode X), both the corresponding database lock and the global lock must be locked in intent exclusive (IX) mode. A single database can simultaneously be locked in IS and IX mode, but an exclusive (X) lock cannot coexist with any other modes, and a shared (S) lock can only coexists with intent shared (IS) locks.

Locks are fair, with reads and writes being queued in order. However, to optimize throughput, when one request is granted, all other compatible requests will be granted at the same time, potentially releasing them before a conflicting request. For example, consider a case in which an X lock was just released, and in which the conflict queue contains the following items:

IS → IS → X → X → S → IS

In strict first-in, first-out (FIFO) ordering, only the first two IS modes would be granted. Instead MongoDB will actually grant all IS and S modes, and once they all drain, it will grant X, even if new IS or S requests have been queued in the meantime. As a grant will always move all other requests ahead in the queue, no starvation of any request is possible.

11.4.2 How granular are locks in MongoDB?

Changed in version 3.0.

Beginning with version 3.0, MongoDB ships with the WiredTiger (page 90) storage engine, which uses optimistic concurrency control for most read and write operations. WiredTiger uses only intent locks at the global, database and collection levels. When the storage engine detects conflicts between two operations, one will incur a write conflict causing MongoDB to transparently retry that operation.

Some global operations, typically short lived operations involving multiple databases, still require a global “instance-wide” lock. Some other operations, such as dropping a collection, still require an exclusive database lock.

The MMAPv1 storage engine uses collection-level locking as of the 3.0 release series, an improvement on earlier versions in which the database lock was the finest-grain lock. Third-party storage engines may either use collection-level locking or implement their own finer-grained concurrency control.

17 See the Wikipedia page on Multiple granularity locking (http://en.wikipedia.org/wiki/Multiple_granularity_locking) for more information.
For example, if you have six collections in a database using the MMAPv1 storage engine and an operation takes a collection-level write lock, the other five collections are still available for read and write operations. An exclusive database lock makes all six collections unavailable for the duration of the operation holding the lock.

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\(^\text{18}\) or Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced\(^\text{19}\)

Specifically, the `locks` document in the output of `serverStatus`, or the `locks` field in the current operation reporting provides insight into the type of locks and amount of lock contention in your `mongod` instance.

To terminate an operation, use `db.killOp()`.

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 operations can also yield locks between individual document modifications in write operations that affect multiple documents like `update()` with the `multi` parameter.

MongoDB’s MMAPv1 (page 91) storage engine 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 into memory. Once data is available in memory, the operation will reacquire the lock to complete the operation.

For storage engines supporting document level concurrency control, yielding is not necessary when accessing storage, as the intent locks held at the global, database and collection level do not block other readers and writers.

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.

\(^\text{18}\)https://cloud.mongodb.com/?jmp=docs
\(^\text{19}\)https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs
### Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Lock Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue a query</td>
<td>Read lock</td>
</tr>
<tr>
<td>Get more data from a cursor</td>
<td>Read lock</td>
</tr>
<tr>
<td>Insert data</td>
<td>Write lock</td>
</tr>
<tr>
<td>Remove data</td>
<td>Write lock</td>
</tr>
<tr>
<td>Update data</td>
<td>Write lock</td>
</tr>
<tr>
<td><strong>Map-reduce</strong></td>
<td>Read lock and write lock, unless operations are specified as non-atomic. Portions of map-reduce jobs can run concurrently.</td>
</tr>
<tr>
<td>Create an index</td>
<td>Building an index in the foreground, which is the default, locks the database for extended periods of time.</td>
</tr>
<tr>
<td><code>db.eval()</code></td>
<td>Write lock. The <code>db.eval()</code> method takes a global write lock while evaluating the JavaScript function. To avoid taking this global write lock, you can use the <code>eval</code> command with <code>nolock: true</code>.</td>
</tr>
<tr>
<td><strong>Deprecated since version 3.0.</strong></td>
<td></td>
</tr>
<tr>
<td><code>eval</code></td>
<td>Write lock. By default, <code>eval</code> command takes a global write lock while evaluating the JavaScript function. If used with <code>nolock: true</code>, the <code>eval</code> 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.</td>
</tr>
<tr>
<td><code>aggregate()</code></td>
<td>Read lock</td>
</tr>
</tbody>
</table>

### 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.createIndex()`, 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 761).

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 320) requires a read lock on the `admin` database for deployments using 2.6 *user credentials* (page 416). 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 its own *locks* (page 758). 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 80)

11.4.13 What isolation guarantees does MongoDB provide?

MongoDB provides the following guarantees in the presence of concurrent read and write operations. These guarantees hold on systems configured with either the MMAPv1 or WiredTiger storage engines.

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, for both the MMAPv1 and WiredTiger storage engines:

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 80)

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 78).

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

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 743) or post your question to the MongoDB User Mailing List20.

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.

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 674). 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 722) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See shardCollection, sh.shardCollection(), the Shard Key (page 674), Deploy a Sharded Cluster (page 691), and SERVER-400021 for more information.

---

20https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
21https://jira.mongodb.org/browse/SERVER-4000
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.

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 176) 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 685), 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 287) section of the UNIX `ulimit` Settings (page 287) 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 690) 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 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.

This is particularly useful for a mongos if you have a client that creates multiple connections and allows them to timeout rather than closing them.

In this case, 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[^2] 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.

### 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 674).

### 11.5.26 What do `moveChunk` commit failed errors mean?

At the end of a *chunk migration* (page 685), 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\(^{23}\) or MongoDB Support\(^{24}\) 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**

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 743) or post your question to the MongoDB User Mailing List\(^{25}\).

**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

\(^{23}\text{http://groups.google.com/group/mongodb-user}\)

\(^{24}\text{https://www.mongodb.org/about/support}\)

\(^{25}\text{https://groups.google.com/forum/?fromgroups#!forum/mongodb-user}\)
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.

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 608)

### 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?

Changed in version 3.0.0: In MongoDB 3.0.0, replica sets can have up to 50 nodes (page 800). Previous versions limited the maximum number of replica set members to 12.

Replica sets are the preferred replication mechanism in MongoDB. However, if your deployment requires more than 50 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 78).

After the default write concern change (page 924), 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 78). See Write Method Acknowledgments (page 857) for more information.

A new protocol for write operations (page 852) 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 572).

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 572)
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 342) 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` is equal to 0, vote in elections. This includes all `delayed` (page 570), `hidden` (page 569) and `secondary-only` (page 567) members. *Arbiters* (page ??) always vote in elections and always have 1 vote.

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 580)

11.6.14 Do hidden members vote in replica set elections?

*Hidden members* (page 569) of replica sets do vote in elections. To exclude a member from voting in an election, change the value of the member's `votes` configuration to 0.

See also:

*Replica Set Elections* (page 580)

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

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 743) or post your question to the MongoDB User Mailing List.

11.7.1 Storage Engine Fundamentals

What is a storage engine?

A storage engine is the part of a database that is responsible for managing how data is stored on disk. Many databases support multiple storage engines, where different engines perform better for specific workloads. For example, one storage engine might offer better performance for read-heavy workloads, and another might support a higher-throughput for write operations.

What will be the default storage engine going forward?

MMAPv1 is the default storage engine in 3.0. With multiple storage engines, you can decide which storage engine is best for your application.

11.7.2 Can you mix storage engines in a replica set?

Yes. You can have a replica set members that use different storage engines.

When designing these multi-storage engine deployments consider the following:

• the oplog on each member may need to be sized differently to account for differences in throughput between different storage engines.
• recovery from backups may become more complex if your backup captures data files from MongoDB: you may need to maintain backups for each storage engine.

11.7.3 WiredTiger Storage Engine

Can I upgrade an existing deployment to a WiredTiger?

Yes. You can upgrade an existing deployment to WiredTiger while the deployment remains available by adding replica set members with the new storage engine and then removing members with the legacy storage engine. See the following sections of the Upgrade MongoDB to 3.0 (page 810) for the complete procedure that you can use to upgrade an existing deployment:

• Change Replica Set Storage Engine to WiredTiger (page 812)
• Change Sharded Cluster Storage Engine to WiredTiger (page 815)

How much compression does WiredTiger provide?

The ratio of compressed data to uncompressed data depends on your data and the compression library used. By default, collection data in WiredTiger use Snappy block compression; zlib compression is also available. Index data use prefix compression by default.

26https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
To what size should I set the WiredTiger cache?

The size of the cache should be sufficient to hold the entire working set for the `mongod`. If the cache does not have enough space to load additional data, WiredTiger evicts pages from the cache to free up space.

To see statistics on the cache and eviction, use the `serverStatus` command. The `cache` field holds the information on the cache and eviction:

```
...  
"wiredTiger" : {  
...,  
"cache" : {  
  "tracked dirty bytes in the cache" : <num>,  
  "bytes currently in the cache" : <num>,  
  "maximum bytes configured" : <num>,  
  "bytes read into cache" : <num>,  
  "bytes written from cache" : <num>,  
  "pages evicted by application threads" : <num>,  
  "checkpoint blocked page eviction" : <num>,  
  "unmodified pages evicted" : <num>,  
  "page split during eviction deepened the tree" : <num>,  
  "modified pages evicted" : <num>,  
  "pages selected for eviction unable to be evicted" : <num>,  
  "pages evicted because they exceeded the in-memory maximum" : <num>,  
  "pages evicted because they had chains of deleted items" : <num>,  
  "failed evictions of pages that exceeded the in-memory maximum" : <num>,  
  "hazard pointer blocked page eviction" : <num>,  
  "internal pages evicted" : <num>,  
  "maximum page size at eviction" : <num>,  
  "eviction server candidate queue empty when topping up" : <num>,  
  "eviction server candidate queue not empty when topping up" : <num>,  
  "eviction server evicting pages" : <num>,  
  "eviction server populating queue, but not evicting pages" : <num>,  
  "eviction server unable to reach eviction goal" : <num>,  
  "pages split during eviction" : <num>,  
  "pages walked for eviction" : <num>,  
  "eviction worker thread evicting pages" : <num>,  
  "in-memory page splits" : <num>,  
  "percentage overhead" : <num>,  
  "tracked dirty pages in the cache" : <num>,  
  "pages currently held in the cache" : <num>,  
  "pages read into cache" : <num>,  
  "pages written from cache" : <num>,  
},  
...  
}``

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

### 11.7.4 MMAPv1 Storage Engine

**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 MMAPv1 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.
How do memory mapped files work?

MongoDB uses memory mapped files for managing and interacting with all data. Memory mapping assigns files to a block of virtual memory with a direct byte-for-byte correlation. MongoDB memory maps data files to memory as it accesses documents. Unaccessed data is not mapped to memory. Once mapped, the relationship between file and memory allows MongoDB to interact with the data in the file as if it were memory.

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

MongoDB preallocates its data files to avoid filesystem fragmentation, and because of this, the size of these files do not necessarily reflect the size of your data. The storage.mmapv1.smallFiles option will reduce the size of these files, which may be useful if you have many small databases on disk.

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 593) 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 308).

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 775).
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 774) 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. If the mongod instance is a member of a replica set, you will need to take the node out of a replica set and restart it as a standalone in order to run repairDatabase.

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 will 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 633) 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 633).

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.

For best performance, the majority of your active set should fit in RAM.

What are page faults?

With the MMAPv1 storage engine, page faults can occur as MongoDB reads from or writes data to parts of its data files that are not currently located in physical memory. In contrast, operating system page faults happen when physical memory is exhausted and pages of physical memory are swapped to disk.
If there is free memory, then the operating system can find the page on disk and load it to memory directly. However, if there is no free memory, the operating system must:

- find a page in memory that is stale or no longer needed, and write the page to disk.
- read the requested page from disk and load it into memory.

This process, 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 214) for more information.

**What is the difference between soft and hard page faults?**

Page faults occur when MongoDB, with the MMAP storage engine, 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.

See [Page Faults](page 214) for more information.

### 11.7.5 Data Storage Diagnostics

**How can I check the size of a collection?**

To view the statistics for a collection, including the data size, use the `db.collection.stats()` method from the mongo shell. The following example issues `db.collection.stats()` for the `orders` collection:

```javascript
db.orders.stats();
```

MongoDB also provides the following methods to return specific sizes for the collection:

- `db.collection.dataSize()` to return data size in bytes for the collection.
- `db.collection.storageSize()` to return allocation size in bytes, including unused space.
- `db.collection.totalSize()` to return the data size plus the index size in bytes.
- `db.collection.totalIndexSize()` to return the index size in bytes.

The following script prints the statistics for each database:

```javascript
db._adminCommand("listDatabases").databases.forEach(function (d) {
    mdb = db.getSiblingDB(d.name);
    printjson(mdb.stats());
});
```

The following script prints the statistics for each collection in each database:

```javascript
db._adminCommand("listDatabases").databases.forEach(function (d) {
    mdb = db.getSiblingDB(d.name);
    mdb.getCollectionNames().forEach(function (c) {
        s = mdb[c].stats();
        printjson(s);
    });
});
```
How can I check the size of indexes for a collection?

To view the size of the data allocated for an index, use the `db.collection.stats()` method and check the `indexSizes` field in the returned document.

How can I get information on the storage use of a database?

The `db.stats()` method in the `mongo` shell returns the current state of the “active” database. For the description of the returned fields, see `dbStats Output`.

11.8 FAQ: Indexes

This document addresses common questions regarding MongoDB indexes.

If you don’t find the answer you’re looking for, check the complete list of FAQs (page 743) or post your question to the MongoDB User Mailing List. See also Indexing Tutorials (page 517).

11.8.1 Should you run `createIndex()` 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 `createIndex()` is usually ok, if an index doesn’t exist because of ongoing administrative work, a call to `createIndex()` may disrupt database availability. Running `createIndex()` can render a replica set inaccessible as the index creation is happening. See Build Indexes on Replica Sets (page 521).

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.

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 550).
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 550), 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 517).

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 507).

If you build a large index without the background option, and if doing so causes the database to stop responding, do one of the following:

- Wait for the index to finish building.
- Kill the current operation (see `db.killOp()`). The partial index will be deleted.

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 550). 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:

```json
{ _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:

```javascript
db.mycollection.find( { attrib: { $elemMatch : { k: "color", v: "blue" } } } )
```

```javascript
db.mycollection.find( { attrib: { $elemMatch : { k: "avail", v: true } } } )
```

11.9 FAQ: MongoDB Diagnostics

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 743) or post your question to the MongoDB User Mailing List29.

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:

```bash
sudo grep mongod /var/log/messages
sudo grep score /var/log/messages
```

11.9.2 Does TCP keepalive time affect MongoDB Deployments?

If you experience socket errors between clients and servers or between members of a sharded cluster or replica set that do not have other reasonable causes, check the TCP keepalive value (e.g. on Linux systems store, the `tcp_keepalive_time` value). A common keepalive period is 7200 seconds (2 hours); however, different distributions and OS X may have different settings.

For MongoDB, you will have better results with shorter keepalive periods, on the order of 120 seconds (two minutes).

If your MongoDB deployment experiences keepalive-related issues, you must alter the keep alive value on all machines hosting MongoDB processes. This includes all machines hosting mongos or mongod servers and all machines hosting client processes that connect to MongoDB.

---

29https://groups.google.com/forum/?fromgroups#!forum/mongodb-user
Note: For non-Linux systems, values greater than or equal to 600 seconds (10 minutes) will be ignored by mongod and mongos. For Linux, values greater than 300 seconds (5 minutes) will be overridden on the mongod and mongos sockets with a maximum of 300 seconds.

On Linux systems:

- To view the keep alive setting, you can use one of the following commands:
  
  ```bash
  sysctl net.ipv4.tcp_keepalive_time
  ```

  Or:

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

  The value is measured in seconds.

- To change the `tcp_keepalive_time` value, you can use one of the following command:
  
  ```bash
  sudo sysctl -w net.ipv4.tcp_keepalive_time=<value>
  ```

  Or:

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

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

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

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

For OS X systems:

- To view the keep alive setting, issue the following command:
  
  ```bash
  sysctl net.inet.tcp.keepinit
  ```

- To change the `net.inet.tcp.keepinit` value, you can use the following command:
  
  ```bash
  sysctl -w net.inet.tcp.keepinit=<value>
  ```

  The above method for setting the TCP keepalive is not persistent; you will need to reset the value each time you reboot or restart a system. See your operating system’s documentation for instructions on setting the TCP keepalive value persistently.

For Windows systems:

- To view the keep alive setting, issue the following command:
  
  ```bash
  reg query HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters /v KeepAliveTime
  ```

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

- To change the `KeepAliveTime` value, use the following command in an Administrator Command Prompt, where `<value>` is expressed in hexadecimal (e.g. 0x0124c0 is 120000):
  
  ```bash
  reg add HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters \ /v KeepAliveTime /d <value>
  ```

  Windows users should consider the Windows Server Technet Article on KeepAliveTime\(^\text{30}\) for more information on setting keep alive for MongoDB deployments on Windows systems.

You will need to restart mongod and mongos servers for new system-wide keepalive settings to take effect.

**11.9.3 What tools are available for monitoring MongoDB?**

The MongoDB Cloud Manager\(^{31}\) and Ops Manager, an on-premise solution available in MongoDB Enterprise Advanced\(^{32}\) include monitoring functionality, which collects data from running MongoDB deployments and provides visualization and alerts based on that data.

For more information, see also the MongoDB Cloud Manager documentation\(^{33}\) and Ops Manager documentation\(^{34}\).

A full list of third-party tools is available as part of the Monitoring for MongoDB (page 187) documentation.

**11.9.4 Memory Diagnostics for the MMAPv1 Storage Engine**

**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 773) 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 a “working set”?**

The working set is the portion of your data that clients access most often.

**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 772) and working set.

---

\(^{31}\)https://cloud.mongodb.com/?jmp=docs

\(^{32}\)https://www.mongodb.com/products/mongodb-enterprise-advanced?jmp=docs

\(^{33}\)https://docs.cloud.mongodb.com/

\(^{34}\)https://docs.opsmanager.mongodb.com/current/application
- The operating system’s cache strategy for LRU (Least Recently Used)
- The impact of journaling (page 308)
- The number or rate of page faults and other MongoDB Cloud Manager gauges to detect when you need more RAM
- 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 773) 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 773), 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 674) and
- *sufficient capacity to support current and future operations* (page 671).

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 685). 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 688) 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 723).

Finally, if your shard key has a low cardinality (page 696), MongoDB may not be able to create sufficient splits among the data.

**Why would one shard receive a disproportioan 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 675).

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 694) 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 782).

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 696) 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 716) that is too short, given the rate of data growth.
  - an uneven distribution of write operations (page 675) 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 716) 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 727) 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 697) 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 675).
Always install the latest, stable version of MongoDB. See *MongoDB Version Numbers* (page 924) 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

*(3.0-series)*

### 12.1.1 Release Notes for MongoDB 3.0

*March 3, 2015*

MongoDB 3.0 is now available. Key features include support for the WiredTiger storage engine, pluggable storage engine API, **SCRAM-SHA-1** authentication mechanism, and improved *explain* functionality.

MongoDB Ops Manager, which includes Automation, Backup, and Monitoring, is now also available. See the [Ops Manager documentation](http://docs.opsmanager.mongodb.com/current/) and the [Ops Manager release notes](http://docs.opsmanager.mongodb.com/current/release-notes/application/) for more information.

### Minor Releases

#### 3.0 Changelog

#### 3.0.4 Changelog

### Security

- **SERVER-18475** authSchemaUpgrade fails when the `system.users` (page 291) contains non **MONGODB-CR** users
- **SERVER-18312** Upgrade PCRE to latest

---

1. [http://docs.opsmanager.mongodb.com/current/](http://docs.opsmanager.mongodb.com/current/)
2. [http://docs.opsmanager.mongodb.com/current/release-notes/application/](http://docs.opsmanager.mongodb.com/current/release-notes/application/)
3. [https://jira.mongodb.org/browse/SERVER-18475](https://jira.mongodb.org/browse/SERVER-18475)
4. [https://jira.mongodb.org/browse/SERVER-18312](https://jira.mongodb.org/browse/SERVER-18312)
Querying

- [SERVER-18364](https://jira.mongodb.org/browse/SERVER-18364) Ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction
- [SERVER-16265](https://jira.mongodb.org/browse/SERVER-16265) Add query details to getmore entry in profiler and db.currentOp()
- [SERVER-15225](https://jira.mongodb.org/browse/SERVER-15225) CachedPlanStage should execute for trial period and re-plan if query performs poorly
- [SERVER-13875](https://jira.mongodb.org/browse/SERVER-13875) ensureIndex() of 2dsphere index breaks after upgrading to 2.6 (with the new createIndex command)

Replication

- [SERVER-18566](https://jira.mongodb.org/browse/SERVER-18566) Primary member can trip fatal assertion if stepping down while running findAndModify op resulting in an upsert
- [SERVER-18511](https://jira.mongodb.org/browse/SERVER-18511) Report upstream progress when initial sync completes
- [SERVER-18409](https://jira.mongodb.org/browse/SERVER-18409) Retry failed heartbeats before marking a node as DOWN
- [SERVER-18326](https://jira.mongodb.org/browse/SERVER-18326) Rollback attempted during initial sync is fatal
- [SERVER-17923](https://jira.mongodb.org/browse/SERVER-17923) Creating/dropping multiple background indexes on the same collection can cause fatal error on secondaries
- [SERVER-17913](https://jira.mongodb.org/browse/SERVER-17913) New primary should log voters at default log level
- [SERVER-17807](https://jira.mongodb.org/browse/SERVER-17807) drain ops before restarting initial sync
- [SERVER-15252](https://jira.mongodb.org/browse/SERVER-15252) Write unit tests of ScatterGatherRunner
- [SERVER-15192](https://jira.mongodb.org/browse/SERVER-15192) Make all logOp listeners rollback-safe
- [SERVER-18190](https://jira.mongodb.org/browse/SERVER-18190) Secondary reads block replication
- [SERVER-18566](https://jira.mongodb.org/browse/SERVER-18566) Primary member can trip fatal assertion if stepping down while running findAndModify op resulting in an upsert

Sharding

- [SERVER-18822](https://jira.mongodb.org/browse/SERVER-18822) Sharded clusters with WiredTiger primaries may lose writes during chunk migration
- [SERVER-18246](https://jira.mongodb.org/browse/SERVER-18246) getmore on secondary in recovery mode can crash mongos
Storage  SERVER-18442 better error message when attempting to change storage engine metadata options

WiredTiger

- SERVER-18647 WiredTiger changes for MongoDB 3.0.4
- SERVER-18646 Avoid WiredTiger checkpointing dead handles
- SERVER-18629 WiredTiger journal system syncs wrong directory
- SERVER-18460 Segfault during eviction under load
- SERVER-18316 Database with WT engine fails to recover after system crash
- SERVER-18315 Throughput drop during transaction pinned phase of checkpoints under WiredTiger
- SERVER-18213 Lots of WriteConflict during multi-upsert with WiredTiger storage engine
- SERVER-18079 Large performance drop with documents > 16k on Windows
- SERVER-17944 WiredTigerRecordStore::truncate spends a lot of time sleeping

HTTP Console  SERVER-18117 Bring back the _replSet page in the html interface

Build and Packaging

- SERVER-18894 OSX SSL builds should use unique filename
- SERVER-18421 Create SSL Builder for OS X
- SERVER-18312 Upgrade PCRE to latest
- SERVER-13596 Support --prefix rpm installation

Internal Code  SERVER-6826 Potential memory leak in ConnectionString::connect

Testing

- SERVER-18318 Disable jsCore_small_oplog suite in Windows
- SERVER-17336 fix core/compact_keeps_indexes.js in a master/slave test configuration
- SERVER-13237 benchRun should use a thread-safe random number generator

^22https://jira.mongodb.org/browse/SERVER-18442
^23https://jira.mongodb.org/browse/SERVER-18647
^24https://jira.mongodb.org/browse/SERVER-18646
^25https://jira.mongodb.org/browse/SERVER-18629
^26https://jira.mongodb.org/browse/SERVER-18460
^27https://jira.mongodb.org/browse/SERVER-18316
^28https://jira.mongodb.org/browse/SERVER-18315
^29https://jira.mongodb.org/browse/SERVER-18213
^30https://jira.mongodb.org/browse/SERVER-18079
^31https://jira.mongodb.org/browse/SERVER-17944
^32https://jira.mongodb.org/browse/SERVER-18117
^33https://jira.mongodb.org/browse/SERVER-18894
^34https://jira.mongodb.org/browse/SERVER-18421
^35https://jira.mongodb.org/browse/SERVER-18312
^36https://jira.mongodb.org/browse/SERVER-13596
^37https://jira.mongodb.org/browse/SERVER-6826
^38https://jira.mongodb.org/browse/SERVER-17336
^39https://jira.mongodb.org/browse/SERVER-13237
3.0.3 Changelog

Security

- SERVER-18290 Adding a read role for a user doesn’t seem to propagate to secondary until restart
- SERVER-18239 `dumpauth.js` uses ambiguous `--db/--collection` args
- SERVER-18169 Regression: Auth enabled arbiter cannot be shutdown using command
- SERVER-18140 Allow `getParameter` to be executed locally against an arbiter in an authenticated replica set
- SERVER-18051 OpenSSL internal error when using SCRAM-SHA1 authentication in FIPS mode
- SERVER-18021 Allow `serverStatus` to be executed locally against an arbiter in an authenticated replica set
- SERVER-17908 Allow `getCmdLineOpts` to be executed locally against an arbiter in an authenticated replica set
- SERVER-17832 Memory leak when `mongod` configured with SSL required and handle insecure connection
- SERVER-17812 `LockPinger` has audit-related GLE failure
- SERVER-17591 Add SSL flag to select supported protocols
- SERVER-16073 Allow disabling SSL Ciphers via hidden flag: `sslCipherConfig`
- SERVER-12235 Don’t require a database read on every new localhost connection when auth is on

Querying

- SERVER-18304 Duplicates on `FindAndModify` with remove option
- SERVER-17815 Plan ranking tie breaker is computed incorrectly

Replication

- SERVER-18211 MongoDB fails to correctly roll back collection creation
- SERVER-17273 Add support for `secondaryCatchupPeriodSecs` to `rs.stepdown()` shell helper

---

41 https://jira.mongodb.org/browse/SERVER-18097
42 https://jira.mongodb.org/browse/SERVER-18290
43 https://jira.mongodb.org/browse/SERVER-18239
44 https://jira.mongodb.org/browse/SERVER-18169
45 https://jira.mongodb.org/browse/SERVER-18140
46 https://jira.mongodb.org/browse/SERVER-18051
47 https://jira.mongodb.org/browse/SERVER-18021
48 https://jira.mongodb.org/browse/SERVER-17908
49 https://jira.mongodb.org/browse/SERVER-17832
50 https://jira.mongodb.org/browse/SERVER-17812
51 https://jira.mongodb.org/browse/SERVER-17591
52 https://jira.mongodb.org/browse/SERVER-16073
53 https://jira.mongodb.org/browse/SERVER-12235
54 https://jira.mongodb.org/browse/SERVER-18304
55 https://jira.mongodb.org/browse/SERVER-17815
56 https://jira.mongodb.org/browse/SERVER-18211
57 https://jira.mongodb.org/browse/SERVER-17273
Sharding

- SERVER-17812\(^{58}\) LockPinger has audit-related GLE failure
- SERVER-17749\(^{59}\) collMod usePowerOf2Sizes fails on mongos
- SERVER-16987\(^{60}\) sh.getRecentMigrations() shows aborted migration as success

Storage

- SERVER-18211\(^{61}\) MongoDB fails to correctly roll back collection creation
- SERVER-18111\(^{62}\) mongod allows user inserts into system.profile collection
- SERVER-17939\(^{63}\) Backport mongo-rocks updates to v3.0 branch
- SERVER-17745\(^{64}\) Improve dirty page estimation in mmapv1 on Windows

WiredTiger

- SERVER-18205\(^{65}\) WiredTiger changes for MongoDB 3.0.3
- SERVER-18192\(^{66}\) Crash running WiredTiger with “cache_resident=true”
- SERVER-18014\(^{67}\) Dropping a collection can block creating a new collection for an extended time under WiredTiger
- SERVER-17907\(^{68}\) B-tree eviction blocks access to collection for extended period under WiredTiger
- SERVER-17892\(^{69}\) Explicitly turn checksum on for all collections/indexes in WiredTiger by default

Indexing

- SERVER-18087\(^{70}\) index_retry.js and index_no_retry.js not checking for presence of “progress” field in currentOp() result
- SERVER-17882\(^{71}\) Update with key too large to index crashes WiredTiger/RockDB secondary

Write Ops

- SERVER-18111\(^{72}\) mongod allows user inserts into system.profile collection

---

\(^{58}\)https://jira.mongodb.org/browse/SERVER-17812
\(^{59}\)https://jira.mongodb.org/browse/SERVER-17749
\(^{60}\)https://jira.mongodb.org/browse/SERVER-16987
\(^{61}\)https://jira.mongodb.org/browse/SERVER-18211
\(^{62}\)https://jira.mongodb.org/browse/SERVER-18111
\(^{63}\)https://jira.mongodb.org/browse/SERVER-17939
\(^{64}\)https://jira.mongodb.org/browse/SERVER-17745
\(^{65}\)https://jira.mongodb.org/browse/SERVER-18205
\(^{66}\)https://jira.mongodb.org/browse/SERVER-18192
\(^{67}\)https://jira.mongodb.org/browse/SERVER-18014
\(^{68}\)https://jira.mongodb.org/browse/SERVER-17907
\(^{69}\)https://jira.mongodb.org/browse/SERVER-17892
\(^{70}\)https://jira.mongodb.org/browse/SERVER-18087
\(^{71}\)https://jira.mongodb.org/browse/SERVER-17882
\(^{72}\)https://jira.mongodb.org/browse/SERVER-18111
Networking

- SERVER-17832 Memory leak when MongoD configured with SSL required and handle insecure connection
- SERVER-17591 Add SSL flag to select supported protocols
- SERVER-16073 Allow disabling SSL Ciphers via hidden flag: sslCipherConfig

Concurrency

- SERVER-18304 duplicates on FindAndModify with remove option
- SERVER-16636 Deadlock detection should check cycles for stability or should be disabled

Geo

- SERVER-17835 Aggregation geoNear deprecated uniqueDocs warning
- SERVER-9220 allow more than two values in the coordinate-array when using 2dsphere index

Aggregation Framework

- SERVER-17835 Aggregation geoNear deprecated uniqueDocs warning

MapReduce

- SERVER-17889 Using eval command to run mapReduce with non-inline “out” option triggers fatal assertion failure

Admin

- SERVER-18290 Adding a read role for a user doesn’t seem to propagate to secondary until restart
- SERVER-18169 Regression: Auth enabled arbiter cannot be shutdown using command
- SERVER-17820 Windows service stop can lead to mongod abrupt termination due to long shutdown time

Build and Packaging

- SERVER-18344 logs should be sent to updated logkeeper server
- SERVER-18299 Use ld wrapper for compiling Enterprise GO tools in RHEL 5
- SERVER-18082 Change smoke.py buildlogger command line options to environment variables

---

https://jira.mongodb.org/browse/SERVER-17832
https://jira.mongodb.org/browse/SERVER-17591
https://jira.mongodb.org/browse/SERVER-16073
https://jira.mongodb.org/browse/SERVER-16636
https://jira.mongodb.org/browse/SERVER-17835
https://jira.mongodb.org/browse/SERVER-9220
https://jira.mongodb.org/browse/SERVER-17889
https://jira.mongodb.org/browse/SERVER-18290
https://jira.mongodb.org/browse/SERVER-18169
https://jira.mongodb.org/browse/SERVER-17820
https://jira.mongodb.org/browse/SERVER-18344
https://jira.mongodb.org/browse/SERVER-18299
https://jira.mongodb.org/browse/SERVER-18082
• SERVER-17730 Parsing of Variables on Windows doesn’t respect windows conventions
• SERVER-17694 support RPATH=value in top-level SConstruct
• SERVER-17465 --use-system-tcmalloc does not support tcmalloc setParameters and extension
• SERVER-17961 THIRD-PARTY-NOTICES.windows’ needs to be updated
• SERVER-17780 Init script sets process ulimit to different value compared to documentation

JavaScript
  • SERVER-17453 warn that db.eval() / eval command is deprecated

Shell
  • SERVER-17951 db.currentOp() fails with read preference set
  • SERVER-17273 Add support for secondaryCatchupPeriodSecs to rs.stepdown shell helper
  • SERVER-16987 sh.getRecentMigrations shows aborted migration as success

Testing
  • SERVER-18302 remove test buildlogger instance
  • SERVER-18262 setup_multiversion_mongodb should retry links download on timeouts
  • SERVER-18239 dumpauth.js uses ambiguous –db/–collection args
  • SERVER-18229 Smoke.py with PyMongo 3.0.1 fails to run certain tests
  • SERVER-18073 Fix smoke.py to work with pymongo 3.0
  • SERVER-17998 Ignore socket exceptions in initial_sync_unsupported_auth_schema.js test
  • SERVER-18293 ASAN tests should run on larger instance size
  • SERVER-17761 RestAdminAccess/NoAdminAccess objects leak at shutdown

3.0.2 Changelog

88https://jira.mongodb.org/browse/SERVER-17730
89https://jira.mongodb.org/browse/SERVER-17694
90https://jira.mongodb.org/browse/SERVER-17465
91https://jira.mongodb.org/browse/SERVER-17961
92https://jira.mongodb.org/browse/SERVER-17780
93https://jira.mongodb.org/browse/SERVER-17453
94https://jira.mongodb.org/browse/SERVER-17951
95https://jira.mongodb.org/browse/SERVER-17273
96https://jira.mongodb.org/browse/SERVER-16987
97https://jira.mongodb.org/browse/SERVER-18302
98https://jira.mongodb.org/browse/SERVER-18262
99https://jira.mongodb.org/browse/SERVER-18239
100https://jira.mongodb.org/browse/SERVER-18229
101https://jira.mongodb.org/browse/SERVER-18073
102https://jira.mongodb.org/browse/SERVER-17998
103https://jira.mongodb.org/browse/SERVER-18293
104https://jira.mongodb.org/browse/SERVER-17761
Security

- SERVER-17719: `mongo` Shell crashes if `-p` is missing and user matches
- SERVER-17705: Fix credentials field inconsistency in HTTP interface
- SERVER-17671: Refuse to complete initial sync from nodes with 2.4-style auth data
- SERVER-17669: Remove auth prompt in webserver when auth is not enabled
- SERVER-17647: Compute BinData length in v8
- SERVER-17529: Can’t list collections when `mongos` is running 3.0 and config servers are running 2.6 and auth is on

Query and Indexing

- SERVER-8188: Configurable idle cursor timeout
- SERVER-17469: 2d nearSphere queries scan entire collection
- SERVER-17642: `WriteConflictException` during background index create

Replication

- SERVER-17677: Replica Set member backtraces sometimes when removed from replica set
- SERVER-17672: `serverStatus` command with `{oplog: 1}` option can trigger segmentation fault in `mongod`
- SERVER-17822: `OpDebug::writeConflicts` should be a 64-bit type

Sharding

- SERVER-17805: `logOp`/`OperationObserver` should always check `shardversion`

Storage

- SERVER-17613: Unable to start `mongod` after unclean shutdown

WiredTiger

- SERVER-17713: WiredTiger using zlib compression can create invalid compressed stream
- SERVER-17642: `WriteConflictException` during background index create
- SERVER-17587: Node crash scenario results in unrecoverable error on subsequent startup under WiredTiger
• SERVER-17562\textsuperscript{122} Invariant failure: \texttt{s->commit_transaction(s, NULL)} resulted in status BadValue 22
• SERVER-17551\textsuperscript{123} mongod fatal assertion after “hazard pointer table full” message
• SERVER-17532\textsuperscript{124} Duplicate key error message does not contain index name anymore
• SERVER-17471\textsuperscript{125} WiredTiger Mutex on Windows can block the server
• SERVER-17382\textsuperscript{126} rc10/wiredTiger multi collection/DB bulk insert slow than rc8 in initial insertion phase
• SERVER-16804\textsuperscript{127} mongod --repair fails because verify() returns EBUSY under WiredTiger

MMAPv1
• SERVER-17616\textsuperscript{128} Removing or inserting documents with large indexed arrays consumes excessive memory
• SERVER-17313\textsuperscript{129} Segfault in BtreeLogic::__insert when inserting into previously-dropped namespace

RocksDB  SERVER-17706\textsuperscript{130} Sync new mongo+rocks changes to v3.0 branch

HTTP Console
• SERVER-17729\textsuperscript{131} Cannot start mongod httpinterface: sockets higher than 1023 not supported
• SERVER-17705\textsuperscript{132} Fix credentials field inconsistency in HTTP interface
• SERVER-17669\textsuperscript{133} Remove auth prompt in webserver when auth is not enabled

Admin
• SERVER-17570\textsuperscript{134} MongoDB 3.0 NT Service shutdown race condition with \texttt{db.serverShutdown()}
• SERVER-17699\textsuperscript{135} “locks” section empty in diagnostic log and profiler output for some operations
• SERVER-17337\textsuperscript{136} RPM Init script breaks with quotes in \texttt{yaml} config file
• SERVER-16731\textsuperscript{137} Remove unused DBPATH init script variable

Networking  SERVER-17652\textsuperscript{138} Cannot start mongod due to “sockets higher than 1023 not being supported”

\textsuperscript{122}https://jira.mongodb.org/browse/SERVER-17562
\textsuperscript{123}https://jira.mongodb.org/browse/SERVER-17551
\textsuperscript{124}https://jira.mongodb.org/browse/SERVER-17532
\textsuperscript{125}https://jira.mongodb.org/browse/SERVER-17471
\textsuperscript{126}https://jira.mongodb.org/browse/SERVER-17382
\textsuperscript{127}https://jira.mongodb.org/browse/SERVER-16804
\textsuperscript{128}https://jira.mongodb.org/browse/SERVER-17616
\textsuperscript{129}https://jira.mongodb.org/browse/SERVER-17313
\textsuperscript{130}https://jira.mongodb.org/browse/SERVER-17706
\textsuperscript{131}https://jira.mongodb.org/browse/SERVER-17729
\textsuperscript{132}https://jira.mongodb.org/browse/SERVER-17705
\textsuperscript{133}https://jira.mongodb.org/browse/SERVER-17669
\textsuperscript{134}https://jira.mongodb.org/browse/SERVER-17570
\textsuperscript{135}https://jira.mongodb.org/browse/SERVER-17699
\textsuperscript{136}https://jira.mongodb.org/browse/SERVER-17337
\textsuperscript{137}https://jira.mongodb.org/browse/SERVER-16731
\textsuperscript{138}https://jira.mongodb.org/browse/SERVER-17652
Testing

- SERVER-17826\(^{139}\) Ignore ismaster exceptions in `initial_sync_unsupported_auth_schema.js` test
- SERVER-17808\(^{140}\) Ensure availability in `initial_sync_unsupported_auth_schema.js` test
- SERVER-17433\(^{141}\) ASAN leak in small oplog suite `write_result.js`

3.0.1 Changelog

Security

- SERVER-17507\(^{142}\) MongoDB3 enterprise AuditLog
- SERVER-17379\(^{143}\) Change “or” to “and” in webserver localhost exception check
- SERVER-16944\(^{144}\) `dbAdminAnyDatabase` should have full parity with `dbAdmin` for a given database
- SERVER-16849\(^{145}\) On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-16452\(^{146}\) Failed login attempts should log source IP address

Querying

- SERVER-17395\(^{147}\) Add FSM tests to stress yielding
- SERVER-17387\(^{148}\) invalid projection for `findAndModify` triggers `fassert()` failure
- SERVER-14723\(^{149}\) Crash during query planning for `geoNear` with multiple 2dsphere indices
- SERVER-17486\(^{150}\) Crash when parsing invalid polygon coordinates

Replication

- SERVER-17515\(^{151}\) `copyDatabase` fails to replicate indexes to secondary
- SERVER-17499\(^{152}\) Using eval command to run `getMore` on aggregation cursor trips fatal assertion
- SERVER-17487\(^{153}\) `cloner dropDups` removes `_id` entries belonging to other records
- SERVER-17302\(^{154}\) consider blacklist in `shouldChangeSyncSource`

---

\(^{139}\) https://jira.mongodb.org/browse/SERVER-17826
^{140}\) https://jira.mongodb.org/browse/SERVER-17808
^{141}\) https://jira.mongodb.org/browse/SERVER-17433
^{142}\) https://jira.mongodb.org/browse/SERVER-17507
^{143}\) https://jira.mongodb.org/browse/SERVER-17379
^{144}\) https://jira.mongodb.org/browse/SERVER-16944
^{145}\) https://jira.mongodb.org/browse/SERVER-16849
^{146}\) https://jira.mongodb.org/browse/SERVER-16452
^{147}\) https://jira.mongodb.org/browse/SERVER-17395
^{148}\) https://jira.mongodb.org/browse/SERVER-17387
^{149}\) https://jira.mongodb.org/browse/SERVER-14723
^{150}\) https://jira.mongodb.org/browse/SERVER-17486
^{151}\) https://jira.mongodb.org/browse/SERVER-17515
^{152}\) https://jira.mongodb.org/browse/SERVER-17499
^{153}\) https://jira.mongodb.org/browse/SERVER-17487
^{154}\) https://jira.mongodb.org/browse/SERVER-17302
Sharding

- SERVER-17398 Deadlock in MigrateStatus::startCommit
- SERVER-17300 Balancer tries to create config.tags index multiple times
- SERVER-16849 On mongos we always invalidate the user cache once, even if no user definitions are changing
- SERVER-5004 balancer should check for stopped between chunk moves in current round

Indexing

- SERVER-17521 improve createIndex validation of empty name
- SERVER-17436 MultiIndexBlock may access deleted collection after recovering from yield

Aggregation Framework

SERVER-17224 Aggregation pipeline with 64MB document can terminate server

Write Ops

- SERVER-17489 in bulk ops, only mark last operation with commit=synchronous
- SERVER-17276 WriteConflictException retry loops needed for collection creation on upsert

Concurrency

- SERVER-17501 Increase journalling capacity limits
- SERVER-17416 Deadlock between MMAP V1 journal lock and oplog collection lock
- SERVER-17395 Add FSM tests to stress yielding

Storage

- SERVER-17515 copyDatabase fails to replicate indexes to secondary
- SERVER-17436 MultiIndexBlock may access deleted collection after recovering from yield
- SERVER-17416 Deadlock between MMAP V1 journal lock and oplog collection lock
- SERVER-17381 Rename rocksExperiment to RocksDB
- SERVER-17369 [Rocks] Fix the calculation of nextPrefix

References:

- SERVER-17398: https://jira.mongodb.org/browse/SERVER-17398
- SERVER-17300: https://jira.mongodb.org/browse/SERVER-17300
- SERVER-16849: https://jira.mongodb.org/browse/SERVER-16849
- SERVER-5004: https://jira.mongodb.org/browse/SERVER-5004
- SERVER-17521: https://jira.mongodb.org/browse/SERVER-17521
- SERVER-17436: https://jira.mongodb.org/browse/SERVER-17436
- SERVER-17224: https://jira.mongodb.org/browse/SERVER-17224
- SERVER-17489: https://jira.mongodb.org/browse/SERVER-17489
- SERVER-17276: https://jira.mongodb.org/browse/SERVER-17276
- SERVER-17501: https://jira.mongodb.org/browse/SERVER-17501
- SERVER-17416: https://jira.mongodb.org/browse/SERVER-17416
- SERVER-17395: https://jira.mongodb.org/browse/SERVER-17395
- SERVER-17381: https://jira.mongodb.org/browse/SERVER-17381
- SERVER-17369: https://jira.mongodb.org/browse/SERVER-17369
• SERVER-17345 WiredTiger -> session.truncate: the start cursor position is after the stop cursor position
• SERVER-17331 RocksDB configuring and monitoring
• SERVER-17323 MMAPV1Journal lock counts are changing during WT run
• SERVER-17319 invariant at shutdown rc9, rc10, rc11 with wiredTiger
• SERVER-17293 Server crash setting wiredTigerEngineRuntimeConfig:"eviction=(threads_max=8)"

WiredTiger
• SERVER-17510 “Didn’t find RecordId in WiredTigerRecordStore” on collections after an idle period
• SERVER-17506 Race between inserts and checkpoints can lose records under WiredTiger
• SERVER-17487 cloner dropDups removes _id entries belonging to other records
• SERVER-17481 WiredTigerRecordStore::validate should call WT_SESSION::verify
• SERVER-17451 WiredTiger unable to start if crash leaves 0-length journal file
• SERVER-17378 WiredTiger’s compact code can return ‘Operation timed out’ error (invariant failure)
• SERVER-17345 WiredTiger -> session.truncate: the start cursor position is after the stop cursor position
• SERVER-17319 invariant at shutdown rc9, rc10, rc11 with wiredTiger

MMAPv1
• SERVER-17501 Increase journalling capacity limits
• SERVER-17416 Deadlock between MMAP V1 journal lock and oplog collection lock
• SERVER-17388 Invariant failure in MMAPv1 when disk full

RocksDB
• SERVER-17381 Rename rocksExperiment to RocksDB
• SERVER-17369 [Rocks] Fix the calculation of nextPrefix
• SERVER-17331 RocksDB configuring and monitoring

---

172https://jira.mongodb.org/browse/SERVER-17345
173https://jira.mongodb.org/browse/SERVER-17331
174https://jira.mongodb.org/browse/SERVER-17323
175https://jira.mongodb.org/browse/SERVER-17319
176https://jira.mongodb.org/browse/SERVER-17293
177https://jira.mongodb.org/browse/SERVER-17510
178https://jira.mongodb.org/browse/SERVER-17506
179https://jira.mongodb.org/browse/SERVER-17487
180https://jira.mongodb.org/browse/SERVER-17481
181https://jira.mongodb.org/browse/SERVER-17451
182https://jira.mongodb.org/browse/SERVER-17378
183https://jira.mongodb.org/browse/SERVER-17345
184https://jira.mongodb.org/browse/SERVER-17319
185https://jira.mongodb.org/browse/SERVER-17388
186https://jira.mongodb.org/browse/SERVER-17369
187https://jira.mongodb.org/browse/SERVER-17331
Shell and Administration

- SERVER-17226191 ‘top’ command with 64MB result document can terminate server
- SERVER-17405192 getLog command masserts when given number
- SERVER-17347193 .explain() should be included in the shell’s DBCollection help

Build and Packaging

- SERVER-17484194 Migrate server MCI config into server repo
- SERVER-17463195 Python error when specifying absolute path to scons cacheDir
- SERVER-17460196 LIBDEPS_v8_SYSLIBDEP typo
- SERVER-14166197 Semantics of the --osx-version-min flag should be improved
- SERVER-17517198 mongodb-org rpm packages no longer “provide” mongo-10gen-server

Logging  SERVER-16452199 Failed login attempts should log source IP address

Platform

- SERVER-17252200 Upgrade PCRE Version from 8.30 to Latest
- SERVER-14166201 Semantics of the --osx-version-min flag should be improved

Internal Code  SERVER-17338202 NULL pointer crash when running copydb against stepped-down 2.6 primary

Testing

- SERVER-17443203 get_replication_info_helper.js should assert.soon rather than assert for log messages
- SERVER-17442204 increase tolerance for shutdown timeout in stepdown.js to fix windows build break
- SERVER-17395205 Add FSM tests to stress yielding

3.0.4 – June 16, 2015

- Fix missed writes with concurrent inserts during chunk migration from shards with WiredTiger primaries:
  SERVER-18822206

---

191https://jira.mongodb.org/browse/SERVER-17226
192https://jira.mongodb.org/browse/SERVER-17405
193https://jira.mongodb.org/browse/SERVER-17347
194https://jira.mongodb.org/browse/SERVER-17484
195https://jira.mongodb.org/browse/SERVER-17463
196https://jira.mongodb.org/browse/SERVER-17460
197https://jira.mongodb.org/browse/SERVER-14166
198https://jira.mongodb.org/browse/SERVER-17517
199https://jira.mongodb.org/browse/SERVER-16452
200https://jira.mongodb.org/browse/SERVER-17252
201https://jira.mongodb.org/browse/SERVER-14166
202https://jira.mongodb.org/browse/SERVER-17338
203https://jira.mongodb.org/browse/SERVER-17443
204https://jira.mongodb.org/browse/SERVER-17442
205https://jira.mongodb.org/browse/SERVER-17395
206https://jira.mongodb.org/browse/SERVER-18822

12.1. Current Stable Release
• Resolve write conflicts with multi-update updates with `upsert=true` with the Wired Tiger Storage engine: SERVER-18213207

• Fix case where secondary reads could block replication: SERVER-18190208

• Improve performance on Windows with WiredTiger and documents larger than 16kb: SERVER-18079209

• Fix issue where WiredTiger data files are not correctly recovered following unexpected system restarts: SERVER-18316210

• All issues closed in 3.0.4211

3.0.3 – May 12, 2015

• Deprecate `db.eval()` and add warnings: SERVER-17453212

• Fix potential for abrupt termination with the Windows service stop operation: SERVER-17802213

• Fix crash caused by update with a `key too large to index` on WiredTiger and RocksDB storage engines: SERVER-17882214

• Remove inconsistent support for `mapReduce` in `eval` environment: SERVER-17889215

• All issues closed in 3.0.3216

3.0.2 – April 9, 2015

• Fix inefficient query plans for `2d $nearSphere`: SERVER-17469217

• Fix problem starting `mongod` during repair operations with WiredTiger: SERVER-17652218 and SERVER-17729219

• Resolved invalid compression stream error with WiredTiger and `zlib` block compression: SERVER-17713220

• Fix memory use issue for inserts into large indexed arrays: SERVER-17616221

• All issues closed in 3.0.2222

3.0.1 – March 17, 2015

• Fixed race condition in WiredTiger between inserts and checkpoints that could result in lost records: SERVER-17506223

207 https://jira.mongodb.org/browse/SERVER-18213
208 https://jira.mongodb.org/browse/SERVER-18190
209 https://jira.mongodb.org/browse/SERVER-18079
210 https://jira.mongodb.org/browse/SERVER-18316
211 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.4%22%20AND%20project%20%3D%20SERVER
212 https://jira.mongodb.org/browse/SERVER-17453
213 https://jira.mongodb.org/browse/SERVER-17802
214 https://jira.mongodb.org/browse/SERVER-17882
215 https://jira.mongodb.org/browse/SERVER-17889
216 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.3%22%20AND%20project%20%3D%20SERVER
217 https://jira.mongodb.org/browse/SERVER-17469
218 https://jira.mongodb.org/browse/SERVER-17652
219 https://jira.mongodb.org/browse/SERVER-17729
220 https://jira.mongodb.org/browse/SERVER-17713
221 https://jira.mongodb.org/browse/SERVER-17616
222 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%223.0.2%22%20AND%20project%20%3D%20SERVER
223 https://jira.mongodb.org/browse/SERVER-17506
• Resolved issue in WiredTiger’s capped collections implementation that caused a server crash: SERVER-17345.
• Fixed issue is initial sync with duplicate _id entries: SERVER-17487.
• Fixed deadlock condition in MMAPv1 between the journal lock and the oplog collection lock: SERVER-17416.
• All issues closed in 3.0.1.

Major Changes

Pluggable Storage Engine API

MongoDB 3.0 introduces a pluggable storage engine API that allows third parties to develop storage engines for MongoDB.

WiredTiger

MongoDB 3.0 introduces support for the WiredTiger storage engine. With the support for WiredTiger, MongoDB now supports two storage engines:

• MMAPv1, the storage engine available in previous versions of MongoDB and the default storage engine for MongoDB 3.0, and
• WiredTiger, available only in the 64-bit versions of MongoDB 3.0.

WiredTiger Usage

WiredTiger is an alternate to the default MMAPv1 storage engine. WiredTiger supports all MongoDB features, including operations that report on server, database, and collection statistics. Switching to WiredTiger, however, requires a change to the on-disk storage format (page 804). For instructions on changing the storage engine to WiredTiger, see the appropriate sections in the Upgrade MongoDB to 3.0 documentation.

MongoDB 3.0 replica sets and sharded clusters can have members with different storage engines; however, performance can vary according to workload. For details, see the appropriate sections in the Upgrade MongoDB to 3.0 documentation.

The WiredTiger storage engine requires the latest official MongoDB drivers. For more information, see WiredTiger and Driver Version Compatibility (page 804).

See also:

Support for touch Command (page 805), WiredTiger Storage Engine (page 90) section in the Storage (page 89) documentation

WiredTiger Configuration

To configure the behavior and properties of the WiredTiger storage engine, see storage.wiredTiger configuration options. You can set WiredTiger options on the command line.

See also:

WiredTiger Storage Engine (page 90) section in the Storage (page 89) documentation

224 https://jira.mongodb.org/browse/SERVER-17345
225 https://jira.mongodb.org/browse/SERVER-17487
226 https://jira.mongodb.org/browse/SERVER-17416
227 https://jira.mongodb.org/issues/?jql=ifixVersion%20%3D%20%223.0.1%22%20AND%20project%20%3D%20SERVER
228 http://wiredtiger.com
229 http://wiredtiger.com
WiredTiger Concurrency and Compression   The 3.0 WiredTiger storage engine provides document-level locking and compression.

By default, WiredTiger compresses collection data using the snappy compression library. WiredTiger uses prefix compression on all indexes by default.

See also:
WiredTiger (page 198) section in the Production Notes (page 197)

MMAPv1 Improvements

MMAPv1 Concurrency Improvement   In version 3.0, the MMAPv1 storage engine adds support for collection-level locking.

MMAPv1 Configuration Changes   To support multiple storage engines, some configuration settings for MMAPv1 have changed. See Configuration File Options Changes (page 804).

MMAPv1 Record Allocation Behavior Changes   MongoDB 3.0 no longer implements dynamic record allocation and deprecates paddingFactor. The default allocation strategy for collections in instances that use MMAPv1 is power of 2 allocation (page 91), which has been improved to better handle large document sizes. In 3.0, the usePowerOf2Sizes flag is ignored, so the power of 2 strategy is used for all collections that do not have noPadding flag set.

For collections with workloads that consist only of inserts or in-place updates (such as incrementing counters), you can disable the power of 2 strategy. To disable the power of 2 strategy for a collection, use the collMod command with the noPadding flag or the db.createCollection() method with the noPadding option.

**Warning:** Do not set noPadding if the workload includes removes or any updates that may cause documents to grow. For more information, see No Padding Allocation Strategy (page 91).

When low on disk space, MongoDB 3.0 no longer errors on all writes but only when the required disk allocation fails. As such, MongoDB now allows in-place updates and removes when low on disk space.

See also:
Dynamic Record Allocation (page 805)

Replica Sets

Increased Number of Replica Set Members

In MongoDB 3.0, replica sets can have up to 50 members. The following drivers support the larger replica sets:

- **C# (.NET) Driver 1.10**
- **Java Driver 2.13**
- **Python Driver (PyMongo) 3.0**
- **Ruby Driver 2.0**
- **Node.JS Driver 2.0**

---

230 The maximum number of voting members remains at 7.
The C, C++, Perl, and PHP drivers, as well as the earlier versions of the Ruby, Python, and Node.JS drivers, discover and monitor replica set members serially, and thus are not suitable for use with large replica sets.

**Replica Set Step Down Behavior Changes**

The process that a *primary* member of a *replica set* uses to step down has the following changes:

- Before stepping down, `replSetStepDown` will attempt to terminate long running user operations that would block the primary from stepping down, such as an index build, a write operation or a map-reduce job.
- To help prevent rollbacks, the `replSetStepDown` will wait for an electable secondary to catch up to the state of the primary before stepping down. Previously, a primary would wait for a secondary to catch up to within 10 seconds of the primary (i.e. a secondary with a replication lag of 10 seconds or less) before stepping down.
- `replSetStepDown` now allows users to specify a `secondaryCatchUpPeriodSecs` parameter to specify how long the primary should wait for a secondary to catch up before stepping down.

**Other Replica Set Operational Changes**

- Initial sync builds indexes more efficiently for each collection and applies oplog entries in batches using threads.
- Definition of `w: “majority”` (page 130) write concern changed to mean majority of *voting* nodes.
- Stronger restrictions on *Replica Set Configuration* (page 652). For details, see *Replica Set Configuration Validation* (page 805).
- For pre-existing collections on secondary members, MongoDB 3.0 no longer automatically builds missing `_id` indexes.

See also:

*Replication Changes* (page 805) in *Compatibility Changes in MongoDB 3.0* (page 804)

**Sharded Clusters**

MongoDB 3.0 provides the following enhancements to sharded clusters:

- Adds a new `sh.removeTagRange()` helper to improve management of sharded collections with tags. The new `sh.removeTagRange()` method acts as a complement to `sh.addTagRange()`.
- Provides a more predictable read preference behavior. *mongos* instances no longer pin connections to members of replica sets when performing read operations. Instead, *mongos* reevaluates *read preferences* (page 588) for every operation to provide a more predictable read preference behavior when read preferences change.
- Provides a new `writeConcern` setting to configure the *write concern* (page 78) of chunk migration operations. You can configure the `writeConcern` setting for the *balancer* (page 714) as well as for *moveChunk* and *cleanupOrphaned* commands.
- Improves visibility of balancer operations. `sh.status()` includes information about the state of the balancer. See `sh.status()` for details.

See also:

*Sharded Cluster Setting* (page 806) in *Compatibility Changes in MongoDB 3.0* (page 804)
Security Improvements

MongoDB 3.0 includes the following security enhancements:

- MongoDB 3.0 adds a new SCRAM-SHA-1 (page 321) challenge-response user authentication mechanism. SCRAM-SHA-1 requires a driver upgrade if your current driver version does not support SCRAM-SHA-1. For the driver versions that support SCRAM-SHA-1, see Upgrade Drivers (page 819).
- Increases restrictions when using the Localhost Exception (page 324) to access MongoDB. For details, see Localhost Exception Changed (page 807).

See also:

Security Changes (page 806)

Improvements

New Query Introspection System

MongoDB 3.0 includes a new query introspection system that provides an improved output format and a finer-grained introspection into both query plan and query execution.

For details, see the new db.collection.explain() method and the new explain command as well as the updated cursor.explain() method.

For information on the format of the new output, see http://docs.mongodb.org/manual/reference/explain-results.

Enhanced Logging

To improve usability of the log messages for diagnosis, MongoDB categorizes some log messages under specific components, or operations, and provides the ability to set the verbosity level for these components. For information, see http://docs.mongodb.org/manual/reference/log-messages.

MongoDB Tools Enhancements

All MongoDB tools except for mongosniff and mongoperf are now written in Go and maintained as a separate project.

- New options for parallelized mongodump and mongorestore. You can control the number of collections that mongorestore will restore at a time with the --numParallelCollections option.
- New options --excludeCollection and --excludeCollectionsWithPrefix for mongodump to exclude collections.
- mongorestore can now accept BSON data input from standard input in addition to reading BSON data from file.
- mongostat and mongotop can now return output in JSON format with the --json option.
- Added configurable write concern to mongoimport, mongorestore, and mongoimport. Use the --writeConcern option.
- mongoimport now allows you to configure the GridFS prefix with the --prefix option so that you can use custom namespaces and store multiple GridFS namespaces in a single database.

See also:

MongoDB Tools Changes (page 806)
Indexes

- Background index builds will no longer automatically interrupt if `dropDatabase`, `drop`, `dropIndexes` operations occur for the database or collection affected by the index builds. The `dropDatabase`, `drop`, and `dropIndexes` commands will still fail with the error message "a background operation is currently running," as in 2.6.

- If you specify multiple indexes to the `createIndexes` command,
  - the command only scans the collection once, and
  - if at least one index is to be built in the foreground, the operation will build all the specified indexes in the foreground.

- For sharded collections, indexes can now cover queries (page 66) that execute against the `mongos` if the index includes the shard key.

See also:

- `Indexes` (page 808) in Compatibility Changes in MongoDB 3.0 (page 804)

Query Enhancements

MongoDB 3.0 includes the following query enhancements:

- For geospatial queries, adds support for “big” polygons for `$geoIntersects` and `$geoWithin` queries. “Big” polygons are single-ringed GeoJSON polygons with areas greater than that of a single hemisphere. See `$geometry`, `$geoIntersects`, and `$geoWithin` for details.

- For `aggregate()`, adds a new `$dateToString` operator to facilitate converting a date to a formatted string.

- Adds the `$eq` query operator to query for equality conditions.

See also:

- 2d Indexes and Geospatial Near Queries (page 808)

Distributions and Supported Versions

Most non-Enterprise MongoDB distributions now include support for TLS/SSL. Previously, only MongoDB Enterprise distributions came with TLS/SSL support included; for non-Enterprise distributions, you had to build MongoDB locally with the `--ssl` flag (i.e. `scons --ssl`).

32-bit MongoDB builds are available for testing, but are not for production use. 32-bit MongoDB builds do not include the WiredTiger storage engine.

MongoDB builds for Solaris do not support the WiredTiger storage engine.

MongoDB builds are available for Windows Server 2003 and Windows Vista (as “64-bit Legacy”), but the minimum officially supported Windows version is Windows Server 2008.

See also:

- Platform Support (page 809), What are the limitations of 32-bit versions of MongoDB? (page 746)
Auditing

*Auditing* (page 330) in MongoDB Enterprise can filter on *any field in the audit message* (page 424), including the fields returned in the *param* (page 425) document. This enhancement, along with the *auditAuthorizationSuccess* parameter, enables auditing to filter on CRUD operations. However, enabling *auditAuthorizationSuccess* to audit of all authorization successes degrades performance more than auditing only the authorization failures.

**Additional Information**

**Changes Affecting Compatibility**

**Compatibility Changes in MongoDB 3.0** The following 3.0 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 3.0* (page 785) for the full list of the 3.0 changes.

**Storage Engine**

**Configuration File Options Changes** With the introduction of additional storage engines in 3.0, some configuration file options have changed:

<table>
<thead>
<tr>
<th>Previous Setting</th>
<th>New Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>storage.journal.commitIntervalMs</td>
<td>storage.mmapv1.journal.commitIntervalMs</td>
</tr>
<tr>
<td>storage.journal.debugFlags</td>
<td>storage.mmapv1.journal.debugFlags</td>
</tr>
<tr>
<td>storage.nsSize</td>
<td>storage.mmapv1.nsSize</td>
</tr>
<tr>
<td>storage.preallocDataFiles</td>
<td>storage.mmapv1.preallocDataFiles</td>
</tr>
<tr>
<td>storage.quota.enforced</td>
<td>storage.mmapv1.quota.enforced</td>
</tr>
<tr>
<td>storage.quota.maxFilesPerDB</td>
<td>storage.mmapv1.quota.maxFilesPerDB</td>
</tr>
<tr>
<td>storage.smallFiles</td>
<td>storage.mmapv1.smallFiles</td>
</tr>
</tbody>
</table>

3.0 *mongod* instances are backward compatible with existing configuration files, but will issue warnings when if you attempt to use the old settings.

**Data Files Must Correspond to Configured Storage Engine** The files in the *dbPath* directory must correspond to the configured storage engine (i.e. *--storageEngine*). *mongod* will not start if *dbPath* contains data files created by a storage engine other than the one specified by *--storageEngine*.

**See also:**

Change Storage Engine to WiredTiger sections in *Upgrade MongoDB to 3.0* (page 810)

**WiredTiger and Driver Version Compatibility** For MongoDB 3.0 deployments that use the WiredTiger storage engine, the following operations return no output when issued in previous versions of the *mongo* shell or drivers:

- `db.getCollectionNames()`
- `db.collection.getIndexes()`
- `show collections`
- `show tables`

Use the 3.0 *mongo* shell or the *3.0 compatible version* (page 808) of the official drivers when connecting to 3.0 *mongod* instances that use WiredTiger. The 2.6.8 *mongo* shell is also compatible with 3.0 *mongod* instances that use WiredTiger.
**db.fsyncLock() is not Compatible with WiredTiger**  
With WiredTiger the `db.fsyncLock()` and `db.fsyncUnlock()` operations cannot guarantee that the data files do not change. As a result, do not use these methods to ensure consistency for the purposes of creating backups.

**Support for touch Command**  
If a storage engine does not support the `touch`, then the `touch` command will return an error.
- The MMAPv1 storage engine supports `touch`.
- The WiredTiger storage engine does not support `touch`.

**Dynamic Record Allocation**  
MongoDB 3.0 no longer supports dynamic record allocation and depreciates `padding-Factor`.

MongoDB 3.0 depreciates the `newCollectionsUsePowerOf2Sizes` parameter such that you can no longer use the parameter to disable the power of 2 sizes allocation for a collection. Instead, use the `collMod` command with the `noPadding` flag or the `db.createCollection()` method with the `noPadding` option. Only set `noPadding` for collections with workloads that consist only of inserts or in-place updates (such as incrementing counters).

**Warning:** Only set `noPadding` to `true` for collections whose workloads have no update operations that cause documents to grow, such as for collections with workloads that are insert-only. For more information, see *No Padding Allocation Strategy* (page 91).

For more information, see *MMAPv1 Record Allocation Behavior Changes* (page 800).

**Replication Changes**

**Replica Set Oplog Format Change**  
MongoDB 3.0 is not compatible with oplog entries generated by versions of MongoDB before 2.2.1. If you upgrade from one of these versions, you must wait for new oplog entries to overwrite all old oplog entries generated by one of these versions before upgrading to 3.0.0 or earlier.

Secondaries may abort if they replay a pre-2.6 oplog with an index build operation that would fail on a 2.6 or later primary.

**Replica Set Configuration Validation**  
MongoDB 3.0 provides a stricter validation of `replica set configuration settings` (page 652) and replica sets invalid replica set configurations.

Stricter validations include:
- Arbiters can only have 1 vote. Previously, arbiters could also have a value of 0 for `votes`. If an arbiter has any value other than 1 for `votes`, you must fix the setting.
- Non-arbiter members can only have value of 0 or 1 for `votes`. If a non-arbiter member has any other value for `votes`, you must fix the setting.
- `_id` in the `Replica Set Configuration` (page 652) must specify the same name as that specified by `--replSet` or `replication.replSetName`. Otherwise, you must fix the setting.
- Disallows 0 for `getLastErrorDefaults` value. If `getLastErrorDefaults` value is 0, you must fix the setting.
- `settings` can only contain the recognized settings. Previously, MongoDB ignored unrecognized settings. If `settings` contains unrecognized settings, you must remove the unrecognized settings.
To fix the settings before upgrading to MongoDB 3.0, connect to the primary and reconfigure your replica set to valid configuration settings.

If you have already upgraded to MongoDB 3.0, you must *downgrade to MongoDB 2.6* (page 821) first and then fix the settings. Once you have reconfigured the replica set, you can re-upgrade to MongoDB 3.0.

**Change of `w: majority` Semantics**  
A write concern with a `w: majority` (page 130) value is satisfied when a majority of the *voting* members replicates a write operation. In previous versions, *majority* referred to a majority of all voting and non-voting members of the set.

**Remove `local.slaves` Collection**  
MongoDB 3.0 removes the `local.slaves` collection that tracked the secondaries’ replication progress. To track the replication progress, use the `rs.status()` method.

**Replica Set State Change**  
The *FATAL* replica set state does not exist as of 3.0.0.

**HTTP Interface**  
The HTTP Interface (i.e. `net.http.enabled`) no longer reports replication data.

**MongoDB Tools Changes**

**Require a Running MongoDB Instance**  
The 3.0 versions of MongoDB tools, `mongodump`, `mongorestore`, `mongoexport`, `mongoimport`, `mongofiles`, and `mongooplog`, must connect to running MongoDB instances and these tools *cannot* directly modify the data files with `--dbpath` as in previous versions. Ensure that you start your `mongod` instance(s) before using these tools.

**Removed Options**

- Removed `--dbpath`, `--journal`, and `--filter` options for `mongodump`, `mongorestore`, `mongoimport`, `mongoexport`, and `bsondump`.
- Removed `--locks` option for `mongotop`.
- Removed `--noobjcheck` option for `bsondump` and `mongorestore`.
- Removed `--csv` option for `mongoexport`. Use the new `--type` option to specify the export format type (`csv` or `json`).

**See also:**

*MongoDB Tools Enhancements* (page 802)

**Sharded Cluster Setting**

**Remove `releaseConnectionsAfterResponse` Parameter**  
MongoDB now always releases connections after response. `releaseConnectionsAfterResponse` parameter is no longer available.

**Security Changes**
MongoDB 2.4 User Model Removed  MongoDB 3.0 completely removes support for the deprecated 2.4 user model. MongoDB 3.0 will exit with an error message if there is user data with the 2.4 schema, i.e. if authSchema version is less than 3. To verify the version of your existing schema, query the system.version collection in the admin database:

```javascript
use admin
db.system.version.find()
```

If you are currently using auth, the query returns the currentVersion of the existing authSchema. If you are not currently using auth, the query will not return any result, and you do not need to take any actions regarding the authSchema version.

If, however, the current authSchema version is less than 3, see Upgrade User Authorization Data to 2.6 Format (page 870) to upgrade to the 2.6 user model before upgrading to 3.0.

After upgrading MongoDB to 3.0 from 2.6, to use the new SCRAM-SHA-1 challenge-response mechanism if you have existing user data, you will need to upgrade the authentication schema a second time. This upgrades the MONGODB-CR user model to SCRAM-SHA-1 user model. See Upgrade to SCRAM-SHA-1 (page 818) for details.

Localhost Exception Changed  In 3.0, the localhost exception changed so that these connections only have access to create the first user on the admin database. In previous versions, connections that gained access using the localhost exception had unrestricted access to the MongoDB instance.

See Localhost Exception (page 324) for more information.

`db.addUser()` Removed  3.0 removes the legacy `db.addUser()` method. Use `db.createUser()` and `db.updateUser()` instead.

TLS/SSL Configuration Option Changes  MongoDB 3.0 introduced new `net.ssl.allowConnectionsWithoutCertificates` configuration file setting and `--sslAllowConnectionsWithoutCertificates` command line option for mongod and mongos. These options replace previous `net.ssl.weakCertificateValidation` and `--sslWeakCertificateValidation` options, which became aliases. Update your configuration to ensure future compatibility.

TLS/SSL Certificates Validation  By default, when running in SSL mode, MongoDB instances will only start if its certificate (i.e. `net.ssl.PemKeyFile`) is valid. You can disable this behavior with the `net.ssl.allowInvalidCertificates` setting or the `--sslAllowInvalidCertificates` command line option.

To start the mongo shell with `--ssl`, you must explicitly specify either the `--sslCAFile` or `--sslAllowInvalidCertificates` option at startup. See TLS/SSL Configuration for Clients (page 346) for more information.

TLS/SSL Certificate Hostname Validation  By default, MongoDB validates the hostnames of hosts attempting to connect using certificates against the hostnames listed in those certificates. In certain deployment situations this behavior may be undesirable. It is now possible to disable such hostname validation without disabling validation of the rest of the certificate information with the `net.ssl.allowInvalidHostnames` setting or the `--sslAllowInvalidHostnames` command line option.

SSLv3 Ciphers Disabled  In light of vulnerabilities in legacy SSL ciphers\[^{231}\], these ciphers have been explicitly disabled in MongoDB. No configuration changes are necessary.

\[^{231}\]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566
**mongo Shell Version Compatibility**  Versions of the `mongo` shell before 3.0 are *not* compatible with 3.0 deployments of MongoDB that enforce access control. If you have a 3.0 MongoDB deployment that requires access control, you must use 3.0 versions of the `mongo` shell.

**HTTP Status Interface and REST API Compatibility**  Neither the HTTP status interface nor the REST API support the `SCRAM-SHA-1` (page 321) challenge-response user authentication mechanism introduced in version 3.0.

**Indexes**

**Remove `dropDups` Option**  The `dropDups` option is no longer available for `createIndex()`, `ensureIndex()`, and `createIndexes`.

**Changes to Restart Behavior during Background Indexing**  For 3.0 `mongod` instances, if a background index build is in progress when the `mongod` process terminates, when the instance restarts the index build will restart as foreground index build. If the index build encounters any errors, such as a duplicate key error, the `mongod` will exit with an error.

To start the `mongod` after a failed index build, use the `storage.indexBuildRetry` or `--noIndexBuildRetry` to skip the index build on start up.

**2d Indexes and Geospatial Near Queries**  For `$near` queries that use a `2d` (page 497) index:

- MongoDB no longer uses a default limit of 100 documents.
- Specifying a `batchSize()` is no longer analogous to specifying a `limit()`.

For `$nearSphere` queries that use a `2d` (page 497) index, MongoDB no longer uses a default limit of 100 documents.

**Driver Compatibility Changes**  Each officially supported driver has released a version that includes support for all new features introduced in MongoDB 3.0. Upgrading to one of these versions is strongly recommended as part of the upgrade process.

A driver upgrade is necessary in certain scenarios due to changes in functionality:

- Use of the `SCRAM-SHA-1` authentication method
- Use of functionality that calls `listIndexes` or `listCollections`

The minimum 3.0-compatible driver versions are:

<table>
<thead>
<tr>
<th>Driver Language</th>
<th>Minimum 3.0-Compatible Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>C232</td>
<td>1.1.0235</td>
</tr>
<tr>
<td>C++234</td>
<td>1.0.0235</td>
</tr>
<tr>
<td>C#236</td>
<td>1.10237</td>
</tr>
<tr>
<td>Java238</td>
<td>2.13239</td>
</tr>
<tr>
<td>Node.js240</td>
<td>1.4.29241</td>
</tr>
<tr>
<td>Perl242</td>
<td>0.708.0.0243</td>
</tr>
<tr>
<td>PHP244</td>
<td>1.6245</td>
</tr>
<tr>
<td>Python246</td>
<td>2.8247</td>
</tr>
<tr>
<td>Motor248</td>
<td>0.4249</td>
</tr>
<tr>
<td>Ruby250</td>
<td>1.12251</td>
</tr>
<tr>
<td>Scala252</td>
<td>2.8.0253</td>
</tr>
</tbody>
</table>

---

232 https://docs.mongodb.org/ecosystem/drivers/
233 https://github.com/mongodb/mongo-c-driver/releases
General Compatibility Changes

**upsert:true with a Dotted _id Query**  When you execute an update() with upsert: true and the query matches no existing document, MongoDB will refuse to insert a new document if the query specifies conditions on the _id field using dot notation (page 171).

This restriction ensures that the order of fields embedded in the _id document is well-defined and not bound to the order specified in the query

If you attempt to insert a document in this way, MongoDB will raise an error.

For example, consider the following update operation. Since the update operation specifies upsert:true and the query specifies conditions on the _id field using dot notation, then the update will result in an error when constructing the document to insert.

```javascript
db.collection.update( { "_id.name": "Robert Frost", "_id.uid": 0 },
    { "categories": ["poet", "playwright"] },
    { upsert: true }
)
```

**Deprecate Access to system.indexes and system.namespaces**  MongoDB 3.0 deprecates direct access to system.indexes and system.namespaces collections. Use the createIndexes and listIndexes commands instead. See also WiredTiger and Driver Version Compatibility (page 804).

**Collection Name Validation**  MongoDB 3.0 more consistently enforces the collection naming restrictions. Ensure your application does not create or depend on invalid collection names.

**Platform Support**  Commercial support is no longer provided for MongoDB on 32-bit platforms (Linux and Windows). Linux RPM and DEB packages are also no longer available. However, binary archives are still available.

**Removed/Deprecated Commands**  The following commands are no longer available in MongoDB 3.0:

- closeAllDatabases
- getoptime
- text

The following commands and methods are deprecated in MongoDB 3.0:

- closeAllDatabases
- getoptime
- text

---

234 https://github.com/mongodb/mongo-cxx-driver
235 https://github.com/mongodb/mongo-cxx-driver/releases
236 https://docs.mongodb.org/ecosystem/drivers/csharp
237 https://github.com/mongodb/mongo-csharp-driver/releases
238 https://docs.mongodb.org/ecosystem/drivers/java
239 https://github.com/mongodb/mongo-java-driver/releases
240 https://docs.mongodb.org/ecosystem/drivers/node-js
241 http://search.cpan.org/dist/MongoDB/
242 https://github.com/mongodb/node-mongodb-native/releases
243 https://docs.mongodb.org/ecosystem/drivers/perl
244 https://docs.mongodb.org/ecosystem/drivers/php
245 http://pecl.php.net/package/mongo
246 https://docs.mongodb.org/ecosystem/drivers/python
247 https://pypi.python.org/pypi/pymongo/
248 https://docs.mongodb.org/ecosystem/drivers/python
249 https://pypi.python.org/pypi/motor/
250 https://docs.mongodb.org/ecosystem/drivers/ruby
251 https://rubygems.org/gems/mongo
252 https://docs.mongodb.org/ecosystem/drivers/scala
253 https://github.com/mongodb/casbah/releases
In addition, you cannot use the now deprecated `eval` command or the `db.eval()` method to invoke `mapReduce` or `db.collection.mapReduce()`.

**Date and Timestamp Comparison Order** MongoDB 3.0 no longer treats the `Timestamp` (page 180) and the `Date` (page 180) data types as equivalent for comparison purposes. Instead, the `Timestamp` (page 180) data type has a higher comparison/sort order (i.e. is “greater”) than the `Date` (page 180) data type. If your application relies on the equivalent comparison/sort order of Date and Timestamp objects, modify your application accordingly before upgrading.

**Server Status Output Change** The `serverStatus` command and the `db.serverStatus()` method no longer return `workingSet`, `indexCounters`, and `recordStats` sections in the output.

**Unix Socket Permissions Change** Unix domain socket file permission now defaults to `0700`. To change the permission, MongoDB provides the `net.unixDomainSocket.filePermissions` setting as well as the `--filePermission` option.

Some changes in 3.0 can affect compatibility (page 804) and may require user actions. For a detailed list of compatibility changes, see *Compatibility Changes in MongoDB 3.0* (page 804).

**Upgrade Process**

**Upgrade MongoDB to 3.0** In the general case, the upgrade from MongoDB 2.6 to 3.0 is a binary-compatible “drop-in” upgrade: shut down the `mongod` instances and replace them with `mongod` instances running 3.0. **However,** before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for upgrading sharded clusters (page 813).

If you need guidance on upgrading to 3.0, MongoDB offers consulting to help ensure a smooth transition without interruption to your MongoDB application.

**Upgrade Recommendations and Checklists** When upgrading, consider the following:

**Upgrade Requirements** To upgrade an existing MongoDB deployment to 3.0, you must be running 2.6. If you’re running a version of MongoDB before 2.6, you must upgrade to 2.6 before upgrading to 3.0. See *Upgrade MongoDB to 2.6* (page 866) for the procedure to upgrade from 2.4 to 2.6. Once upgraded to MongoDB 2.6, you cannot downgrade to any version earlier than MongoDB 2.4.

If your existing MongoDB deployment is already running with authentication and authorization, your user data model `authSchema` must be at least version 3. To verify the version of your existing `authSchema`, see *MongoDB 2.4 User Model Removed* (page 807). To upgrade your `authSchema` version, see *Upgrade User Authorization Data to 2.6 Format* (page 870) for details.

**Preparedness** Before upgrading MongoDB, always test your application in a staging environment before deploying the upgrade to your production environment.

Some changes in MongoDB 3.0 require manual checks and intervention. Before beginning your upgrade, see the *Compatibility Changes in MongoDB 3.0* (page 804) document to ensure that your applications and deployments are compatible with MongoDB 3.0. Resolve the incompatibilities in your deployment before starting the upgrade.

---

254[https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade](https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade)
Downgrade Limitations Once upgraded to MongoDB 3.0, you cannot downgrade to a version lower than 2.6.5. If you upgrade to 3.0 and have run authSchemaUpgrade, you cannot downgrade to 2.6 without disabling --auth or restoring a pre-upgrade backup, as authSchemaUpgrade discards the MONGODB-CR credentials used in 2.6. See Upgrade Existing MONGODB-CR Users to Use SCRAM-SHA-1 (page 817).

Upgrade MongoDB Processes

Upgrade Standalone mongod Instance to MongoDB 3.0 The following steps outline the procedure to upgrade a standalone mongod from version 2.6 to 3.0. To upgrade from version 2.4 to 3.0, upgrade to version 2.6 (page 866) first, and then use the following procedure to upgrade from 2.6 to 3.0.

Upgrade Binaries If you installed MongoDB from the MongoDB apt, yum, or zypper repositories, you should upgrade to 3.0 using your package manager. Follow the appropriate installation instructions (page 6) for your Linux system. This will involve adding a repository for the new release, then performing the actual upgrade. Otherwise, you can manually upgrade MongoDB:

Step 1: Download 3.0 binaries. Download binaries of the latest release in the 3.0 series from the MongoDB Download Page. See Install MongoDB (page 5) for more information.

Step 2: Replace 2.6 binaries. Shut down your mongod instance. Replace the existing binary with the 3.0 mongod binary and restart mongod.

Change Storage Engine to WiredTiger To change storage engine to WiredTiger, you will need to manually export and upload the data using mongodump and mongorestore.

Step 1: Start 3.0 mongod. Ensure that the 3.0 mongod is running with the default MMAPv1 engine.

Step 2: Export the data using mongodump.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See mongodump for available options.

Step 3: Create data directory for WiredTiger. Create a new data directory for WiredTiger. Ensure that the user account running mongod has read and write permissions for the new directory.

mongod with WiredTiger will not start with data files created with a different storage engine.

Step 4: Restart the mongod with WiredTiger. Restart the 3.0 mongod, specifying wiredTiger as the --storageEngine and the newly created data directory for WiredTiger as the --dbpath. Specify additional options as appropriate.

```
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>
```

You can also specify the options in a configuration file. To specify the storage engine, use the new storage.engine setting.

---

255 http://www.mongodb.org/downloads?jmp=docs
Step 5: Upload the exported data using `mongorestore`.

`mongorestore <exportDataDestination>`

Specify additional options as appropriate. See `mongorestore` for available options.

Upgrade a Replica Set to 3.0

Prerequisites

- If the oplog contains entries generated by versions of MongoDB that precede version 2.2.1, you must wait for the entries to be overwritten by later versions before you can upgrade to MongoDB 3.0. For more information, see `Replica Set Oplog Format Change` (page 805).

- `Stricter validation in MongoDB 3.0` (page 805) of replica set configuration may invalidate previously-valid replica set configuration, preventing replica sets from starting in MongoDB 3.0. For more information, see `Replica Set Configuration Validation` (page 805).

- All replica set members must be running version 2.6 before you can upgrade them to version 3.0. To upgrade a replica set from an earlier MongoDB version, `upgrade all members of the replica set to version 2.6` (page 866) first, and then follow the procedure to upgrade from MongoDB 2.6 to 3.0.

Upgrade Binaries

You can upgrade from MongoDB 2.6 to 3.0 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 2.6 binary with the 3.0 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 580). `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 3.0 binary and start the new instance.

Replica set failover is not instant and will render the set unavailable to accept writes until the failover process completes. This may take 30 seconds or more: schedule the upgrade procedure during a scheduled maintenance window.

Change Replica Set Storage Engine to WiredTiger

In MongoDB 3.0, replica sets can have members with different storage engines. As such, you can update members to use the WiredTiger storage engine in a rolling manner. Before changing all the members to use WiredTiger, you may prefer to run with mixed storage engines for some period. However, performance can vary according to workload.

**Note:** Before enabling the new WiredTiger storage engine, ensure that all replica set/sharded cluster members are running at least MongoDB version 2.6.8, and preferably version 3.0.0 or newer.

To change the storage engine to WiredTiger for an existing secondary replica set member, remove the member’s data and perform an `initial sync` (page 633):
Step 1: Shutdown the secondary member. Stop the mongod instance for the secondary member.

Step 2: Prepare data directory for WiredTiger. mongod with WiredTiger will not start if the --dbpath directory contains data files created with a different storage engine.

For the stopped secondary member, either delete the content of the data directory or create a new data directory. If creating a new directory, ensure that the user account running mongod has read and write permissions for the new directory.

Step 3: Restart the secondary member with WiredTiger. Restart the 3.0 mongod, specifying wiredTiger as the --storageEngine and the data directory for WiredTiger as the --dbpath. Specify additional options as appropriate for the member.

mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath>

Since no data exists in the --dbpath, 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.

You can also specify the options in a configuration file. To specify the storage engine, use the new storage.engine setting.

To update all members of the replica set to use WiredTiger, update the secondary members first. Then step down the primary, and update the stepped-down member.

Upgrade a Sharded Cluster to 3.0 Only upgrade sharded clusters to 3.0 if all members of the cluster are currently running instances of 2.6. The only supported upgrade path for sharded clusters running 2.4 is via 2.6. The upgrade process checks all components of the cluster and will produce warnings if any component is running version 2.4.

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 736) for a complete list of sharding commands. Note, however, that not all commands on the Sharding Reference (page 736) 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 684) in the sharded cluster, as described in Disable the Balancer (page 717).
**Step 2: Upgrade the cluster’s meta data.** Start a single 3.0 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 3.0, or if you need to avoid reconfiguring a production mongos instance, you can use a new 3.0 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 3.0 mongos will output informational log messages.

```
<timestamp> I SHARDING [mongoMain] MongoS version 3.0.0 starting: ...
... <timestamp> I SHARDING [mongoMain] starting upgrade of config server from v5 to v6
<timestamp> I SHARDING [mongoMain] starting next upgrade step from v5 to v6
<timestamp> I SHARDING [mongoMain] about to log new metadata event: ...
<timestamp> I SHARDING [mongoMain] checking that version of host ... is compatible with 2.6 ...
<timestamp> I SHARDING [mongoMain] upgrade of config server to v6 successful ...
<timestamp> I SHARDING [mongoMain] distributed lock 'configUpgrade/...' unlocked.
<timestamp> I - [mongoMain] Config database is at version v6
```

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:

```
<timestamp> I SHARDING [mongoMain] upgrade of config server to v6 successful ...
<timestamp> I - [mongoMain] Config database is at version v6
```

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 3.0.** Upgrade and restart without the --upgrade option the other mongos instances in the sharded cluster.

After you have successfully upgraded all mongos instances, you can proceed to upgrade the other components in your sharded cluster.

**Warning:** Do not upgrade the mongod instances until after you have upgraded all the mongos instances.
Step 5: Upgrade the config servers. After you have successfully upgraded all mongos instances, upgrade all 3 mongod config server instances, leaving the first config server listed in the mongos --configdb argument to upgrade last.

Step 6: Upgrade the shards. Upgrade each shard, one at a time, upgrading the mongod secondaries before running replSetStepDown and upgrading the primary of each shard.

Step 7: Re-enable the balancer. Once the upgrade of sharded cluster components is complete, Re-enable the balancer (page 718).

Change Sharded Cluster Storage Engine to WiredTiger For a sharded cluster in MongoDB 3.0, you can choose to update the shards to use WiredTiger storage engine and have the config servers use MMAPv1. If you update the config servers to use WiredTiger, you must update all three config servers to use WiredTiger.

Considerations Before enabling the new WiredTiger storage engine, ensure that all replica set/sharded cluster members are running at least MongoDB version 2.6.8, and preferably version 3.0.0 or newer.

Note: You may safely continue to use MMAPv1 for the config servers even if the rest of your cluster is using WiredTiger.

Change Shards to Use WiredTiger To change the storage engine for the shards to WiredTiger, refer to the procedures in Change Storage Engine to WiredTiger for replica sets (page 812) and Change Storage Engine to WiredTiger for standalone mongod (page 811) as appropriate for your shards.

Change Config Server to Use WiredTiger For a sharded cluster in MongoDB 3.0, you may continue to use MMAPv1 for the config servers even if the shards use WiredTiger; if, however, you decide to change the config servers to use the WiredTiger storage engine, you must change all three config servers to use WiredTiger.

Note: During this process, only two config servers will be running at any given time to ensure that the sharded cluster’s metadata is read only.

As a reminder, you may safely continue to use MMAPv1 for the config servers even if the rest of your cluster is using WiredTiger.

Step 1: Disable the Balancer. Turn off the balancer (page 684) in the sharded cluster, as described in Disable the Balancer (page 717).

Step 2: Stop the last config server listed in the mongos’ configDB setting.

Step 3: Export data of the second config server listed in the mongos’ configDB setting.

mongodump --out <exportDataDestination>

Specify additional options as appropriate, such as username and password if running with authorization enabled. See mongodump for available options.
Step 4: **For the second config server, create a new data directory for WiredTiger.** Ensure that the user account running mongod has read and write permissions for the new directory.

mongod with WiredTiger will not start if the `--dbpath` directory contains data files created with a different storage engine.

Step 5: **Restart the second config server with WiredTiger.** Specify `wiredTiger` as the `--storageEngine` and the newly created WiredTiger data directory as the `--dbpath` as well as any additional options as appropriate.

```bash
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath> --configsvr
```

You can also specify the options in a configuration file. To specify the storage engine, use the new `storage.engine` setting.

Step 6: **Upload the exported data using mongorestore to the second config server.**

```bash
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` for available options.

Step 7: **Shut down the second config server.**

Step 8: **Restart the third config server.**

Step 9: **Export data of the third config server.**

```bash
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` for available options.

Step 10: **For the third config server, create a new data directory for WiredTiger.** Ensure that the user account running mongod has read and write permissions for the new directory.

mongod with WiredTiger will not start if the `--dbpath` directory contains data files created with a different storage engine.

Step 11: **Restart the third config server with WiredTiger.** Specify `wiredTiger` as the `--storageEngine` and the newly created WiredTiger data directory as the `--dbpath` as well as any additional options as appropriate.

```bash
mongod --storageEngine wiredTiger --dbpath <newWiredTigerDBPath> --configsvr
```

You can also specify the options in a configuration file. To specify the storage engine, use the new `storage.engine` setting.

Step 12: **Upload the exported data using mongorestore to the third config server.**

```bash
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` for available options.

Step 13: **Export data of the first config server listed in the mongos’ configDB setting.**
Step 14: For the first config server, create data directory for WiredTiger. Ensure that the user account running mongod has read and write permissions for the new directory.

mongod with WiredTiger will not start if the --dbpath directory contains data files created with a different storage engine.

Step 15: Restart the first config server with WiredTiger. Specify wiredTiger as the --storageEngine and the newly created WiredTiger data directory as the --dbpath as well as any additional options as appropriate.

You can also specify the options in a configuration file. To specify the storage engine, use the new storage.engine setting.

Step 16: Upload the exported data using mongorestore to the first config server.

You can also specify the options in a configuration file. To specify the storage engine, use the new storage.engine setting.

Step 17: Enable writes to the sharded cluster’s metadata. Restart the second config server, specifying WiredTiger as the --storageEngine and the newly created WiredTiger data directory as the --dbpath. Specify additional options as appropriate.

You can also specify the options in a configuration file. To specify the storage engine, use the new storage.engine setting. Once all three config servers are up, the sharded cluster’s metadata is available for writes.

Step 18: Re-enable the balancer. Once all three config servers are up and running with WiredTiger, Re-enable the balancer (page 718).

Upgrade Existing MONGODB–CR Users to Use SCRAM–SHA–1 After upgrading the binaries, see Upgrade to SCRAM-SHA-1 (page 818) for details on SCRAM-SHA-1 upgrade scenarios.

General Upgrade Procedure Except as described on this page, moving between 2.6 and 3.0 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 225) for alternate methods of stopping a mongod instance.
Step 2: Start the new mongod instance. Ensure you start the 3.0 mongod with the same dbPath:

```
mongod --dbpath /var/mongod/data
```

Replace /var/mongod/data with your MongoDB dbPath.

Upgrade to SCRAM-SHA-1

Overview MongoDB 3.0 includes support for the SCRAM-SHA-1 (page 321) challenge-response user authentication mechanism, which changes how MongoDB uses and stores user credentials.

For deployments that already contain user authentication data, to use the SCRAM-SHA-1 mechanism, you must upgrade the authentication schema in addition to upgrading the MongoDB processes.

You may, alternatively, opt to continue to use the MONGODB-CR challenge-response mechanism and skip this upgrade. See Upgrade Scenarios (page 818) for details.

Upgrade Scenarios The following scenarios are possible when upgrading from 2.6 to 3.0:

Continue to Use MONGODB-CR If you are upgrading from a 2.6 database with existing user authentication data, to continue to use MONGODB-CR for existing challenge-response users, no upgrade to the existing user data is required. However, new challenge-response users created in 3.0 will use the following authentication mechanism:

- If you populated MongoDB 3.0 user data by importing the 2.6 user authentication data, including user data, new challenge-response users created in MongoDB 3.0 will use SCRAM-SHA1.
- If you run MongoDB 3.0 binary against the 2.6 data files, including the user authentication data files, new challenge-response users created in MongoDB 3.0 will continue to use the MONGODB-CR.

You can execute the upgrade to SCRAM-SHA-1 at any point in the future.

Use SCRAM-SHA-1

- If you are starting with a new 3.0 installation without any users or upgrading from a 2.6 database that has no users, to use SCRAM-SHA-1, no user data upgrade is required. All newly created users will have the correct format for SCRAM-SHA-1.
- If you are upgrading from a 2.6 database with existing user data, to use SCRAM-SHA-1, follow the steps in Upgrade 2.6 MONGODB-CR Users to SCRAM-SHA-1 (page 820).

Important: Before you attempt any upgrade, familiarize yourself with the Considerations (page 818) as the upgrade to SCRAM-SHA-1 is irreversible short of restoring from backups.

Recommendation SCRAM-SHA-1 represents a significant improvement in security over MONGODB-CR, the previous default authentication mechanism: you are strongly urged to upgrade. For advantages of using SCRAM-SHA-1, see SCRAM-SHA-1 (page 321).

Considerations

Backwards Incompatibility The procedure to upgrade to SCRAM-SHA-1 discards the MONGODB-CR credentials used by 2.6. As such, the procedure is irreversible, short of restoring from backups.

The procedure also disables MONGODB-CR as an authentication mechanism.
Upgrade Binaries  Before upgrading the authentication model, you should first upgrade MongoDB binaries to 3.0. For sharded clusters, ensure that all cluster components are 3.0.

Upgrade Drivers  You must upgrade all drivers used by applications that will connect to upgraded database instances to version that support SCRAM–SHA–1. The minimum driver versions that support SCRAM–SHA–1 are:

<table>
<thead>
<tr>
<th>Driver Language</th>
<th>Version</th>
</tr>
</thead>
</table>
| C
256 |
| C++
258 |
| C#
260 |
| Java
262 |
| Node.js
264 |
| Perl
266 |
| PHP
268 |
| Python
270 |
| Motor
272 |
| Ruby
274 |
| Scala
276 |

1.1.0
257 |
1.0.0
259 |
1.10
261 |
2.13
263 |
1.4.29
265 |
0.708.0
267 |
1.6
269 |
2.8
271 |
0.4
273 |
1.12
275 |
2.8.0
277 |

See the MongoDB Drivers Page for links to download upgraded drivers.

Requirements  To upgrade the authentication model, you must have a user in the admin database with the role userAdminAnyDatabase (page 412).

Timing  Because downgrades are more difficult after you upgrade the user authentication model, once you upgrade the MongoDB binaries to version 3.0, allow your MongoDB deployment to run for a day or two before following this procedure.

This allows 3.0 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 2.6.

If you decide to upgrade the user authentication 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 authentication upgrade command.

256https://docs.mongodb.org/ecosystem/drivers/c
257https://github.com/mongodb/mongo-c-driver/releases
258https://github.com/mongodb/mongo-cxx-driver/releases
259https://github.com/mongodb/mongo-c-driver/releases
260https://docs.mongodb.org/ecosystem/drivers/csharp
261https://github.com/mongodb/mongo-csharp-driver/releases
262https://github.com/mongodb/mongo-java-driver/releases
263https://github.com/mongodb/mongo-java-driver/releases
264https://github.com/mongodb/mongo-java-driver/releases
265https://github.com/mongodb/node-mongodb-native/releases
266https://github.com/mongodb/mongo-c-driver/releases
267http://search.cpan.org/dist/MongoDB/
268https://docs.mongodb.org/ecosystem/drivers/php
269http://pecl.php.net/package/mongo
270https://docs.mongodb.org/ecosystem/drivers/python
271https://pypi.python.org/pypi/pymongo/
272https://pypi.python.org/pypi/motor/
273https://pypi.python.org/pypi/motor/
274https://docs.mongodb.org/ecosystem/drivers/ruby
275https://rubygems.org/gems/mongo
276https://docs.mongodb.org/ecosystem/drivers/scala
277https://github.com/mongodb/casbah/releases
278https://docs.mongodb.org/ecosystem/drivers
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 one *mongos* instance and run the upgrade procedure to upgrade the cluster’s authentication data. By default, the procedure will upgrade the authentication data of the shards as well.

To override this behavior, run `authSchemaUpgrade` with the `upgradeShards: false` option. 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 670). Instead, perform the upgrade process using one *mongos* instance to interact with the config database.

---

Upgrade 2.6 MONGODB-CR Users to SCRAM-SHA-1

**Warning:** The procedure to upgrade to SCRAM-SHA-1 **discards** the MONGODB-CR credentials used by 2.6. As such, the procedure is **irreversible**, short of restoring from backups. The procedure also disables MONGODB-CR as an authentication mechanism.

---

**Important:** To use the SCRAM-SHA-1 authentication mechanism, a driver upgrade is **necessary** if your current driver version does not support SCRAM-SHA-1. See *required driver versions* (page 819) for details.

---

**Step 1: Connect to the MongoDB instance.** Connect and authenticate to the *mongod* instance for a single deployment, the primary *mongod* for a replica set, or a *mongos* for a sharded cluster as an *admin* database user with the role *userAdminAnyDatabase* (page 412).

**Step 2: Upgrade authentication schema.** Use the `authSchemaUpgrade` command in the *admin* database to update the user data using the *mongo* shell.

**Run authSchemaUpgrade command.**

```
db.adminCommand({authSchemaUpgrade: 1});
```

In case of error, you may safely rerun the `authSchemaUpgrade` command.

**Sharded cluster authSchemaUpgrade consideration.** For a sharded cluster without shard local users (page 324), `authSchemaUpgrade` will, by default, upgrade the authorization data of the shards as well, completing the upgrade.

You can, however, override this behavior by including `upgradeShards: false` in the command, as in the following example:

```
db.adminCommand({
   authSchemaUpgrade: 1, upgradeShards: false
});
```

If you override the default behavior or your cluster has shard local users, 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** After this procedure is complete, all users in the database will have SCRAM-SHA-1-style credentials, and any subsequently-created users will also have this type of credentials.
Downgrading MongoDB from 3.0

Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the **Downgrade Recommendations and Checklist** (page 821) and the procedure for downgrading **sharded clusters** (page 823).

**Downgrade Recommendations and Checklist** When downgrading, consider the following:

**Downgrade Path** Once upgraded to MongoDB 3.0, you **cannot** downgrade to a version lower than **2.6.5**.

**Important:** If you upgrade to MongoDB 3.0 and have run `authSchemaUpgrade`, you **cannot** downgrade to the 2.6 series without disabling `--auth`.

**Procedures** Follow the downgrade procedures:

- To downgrade sharded clusters, see **Downgrade a 3.0 Sharded Cluster** (page 823).
- To downgrade replica sets, see **Downgrade a 3.0 Replica Set** (page 822).
- To downgrade a standalone MongoDB instance, see **Downgrade a Standalone mongod Instance** (page 821).

**Note:** Optional. Consider compacting collections after downgrading. Otherwise, older versions will not be able to reuse free space regions larger than 2MB created while running 3.0. This can result in wasted space but no data loss following the downgrade.

**Downgrade MongoDB Processes**

**Downgrade a Standalone mongod Instance** If you have changed the storage engine to **WiredTiger**, change the storage engine to MMAPv1 before downgrading to 2.6.

**Change Storage Engine to MMAPv1** To change storage engine to MMAPv1 for a standalone mongod instance, you will need to manually export and upload the data using `mongodump` and `mongorestore`.

**Step 1:** Ensure 3.0 mongod is running with WiredTiger.

**Step 2:** Export the data using `mongodump`.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` for available options.

**Step 3:** Create data directory for MMAPv1. Create a new data directory for MMAPv1. Ensure that the user account running mongod has read and write permissions for the new directory.
Step 4: Restart the `mongod` with MMAPv1. Restart the 3.0 `mongod`, specifying the newly created data directory for MMAPv1 as the `--dbpath`. You do not have to specify `--storageEngine` as MMAPv1 is the default.

```
mongod --dbpath <newMMAPv1DBPath>
```

Specify additional options as appropriate.

Step 5: Upload the exported data using `mongorestore`.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` for available options.

**Downgrade Binaries** The following steps outline the procedure to downgrade a standalone `mongod` from version 3.0 to 2.6.

Once upgraded to MongoDB 3.0, you cannot downgrade to a version lower than 2.6.5.

**Step 1: Download 2.6 binaries.** Download binaries of the latest release in the 2.6 series from the MongoDB Download Page. See Install MongoDB (page 5) for more information.

**Step 2: Replace with 2.6 binaries.** Shut down your `mongod` instance. Replace the existing binary with the 2.6 `mongod` binary and restart `mongod`.

**Downgrade a 3.0 Replica Set** If you have changed the storage engine to WiredTiger, change the storage engine to MMAPv1 before downgrading to 2.6.

**Change Storage Engine to MMAPv1** You can update members to use the MMAPv1 storage engine in a rolling manner.

**Note:** When running a replica set with mixed storage engines, performance can vary according to workload.

To change the storage engine to MMAPv1 for an existing secondary replica set member, remove the member’s data and perform an initial sync (page 633):

**Step 1: Shutdown the secondary member.** Stop the `mongod` instance for the secondary member.

**Step 2: Prepare data directory for MMAPv1.** Prepare `--dbpath` directory for initial sync.

For the stopped secondary member, either delete the content of the data directory or create a new data directory. If creating a new directory, ensure that the user account running `mongod` has read and write permissions for the new directory.

**Step 3: Restart the secondary member with MMAPv1.** Restart the 3.0 `mongod`, specifying the MMAPv1 data directory as the `--dbpath`. Specify additional options as appropriate for the member. You do not have to specify `--storageEngine` since MMAPv1 is the default.

```
mongod --dbpath <preparedMMAPv1DBPath>
```

Note: [https://www.mongodb.org/downloads](https://www.mongodb.org/downloads)
Since no data exists in the --dbpath, 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.

Repeat for the remaining the secondary members. Once all the secondary members have switched to MMAPv1, step down the primary, and update the stepped-down member.

**Downgrade Binaries**   Once upgraded to MongoDB 3.0, you cannot downgrade to a version lower than 2.6.5.

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 secondary members of the replica set.**   Downgrade each secondary member of the replica set, one at a time:

1. Shut down the mongod. See *Stop mongod Processes* (page 225) for instructions on safely terminating mongod processes.
2. Replace the 3.0 binary with the 2.6 binary and restart.
3. Wait for the member to recover to SECONDARY state before downgrading 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 580) procedure.

```javascript
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 2.6 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 3.0 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()
- 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 736) for a complete list of sharding commands. Note, however, that not all commands on the Sharding Reference (page 736) page modifies the cluster meta-data.

Change Storage Engine to MMAPv1  If you have changed the storage engine to WiredTiger, change the storage engine to MMAPv1 before downgrading to 2.6.

Change Shards to Use MMAPv1  To change the storage engine to MMAPv1, refer to the procedure in Change Storage Engine to MMAPv1 for replica set members (page 822) and Change Storage Engine to MMAPv1 for standalone mongod (page 821) as appropriate for your shards.

Change Config Servers to Use MMAPv1  Note: During this process, only two config servers will be running at any given time to ensure that the sharded cluster’s metadata is read only.

---

Step 1: Disable the Balancer.  Turn off the balancer (page 684) in the sharded cluster, as described in Disable the Balancer (page 717).

Step 2: Stop the last config server listed in the mongos’ configDB setting.

Step 3: Export data of the second config server listed in the mongos’ configDB setting.

```bash
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See mongodump for available options.

Step 4: For the second config server, create a new data directory for MMAPv1.  Ensure that the user account running mongod has read and write permissions for the new directory.

Step 5: Restart the second config server with MMAPv1.  Specify the newly created MMAPv1 data directory as the --dbpath as well as any additional options as appropriate.

```bash
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 6: Upload the exported data using mongorestore to the second config server.

```bash
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See mongorestore for available options.

Step 7: Shut down the second config server.

Step 8: Restart the third config server.
Step 9: Export data of the third config server.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` for available options.

Step 10: For the third config server, create a new data directory for MMAPv1. Ensure that the user account running `mongod` has read and write permissions for the new directory.

Step 11: Restart the third config server with MMAPv1. Specify the newly created MMAPv1 data directory as the `--dbpath` as well as any additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 12: Upload the exported data using `mongorestore` to the third config server.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` for available options.

Step 13: Export data of the first config server listed in the `mongos` `configDB` setting.

```
mongodump --out <exportDataDestination>
```

Specify additional options as appropriate, such as username and password if running with authorization enabled. See `mongodump` for available options.

Step 14: For the first config server, create data directory for MMAPv1. Ensure that the user account running `mongod` has read and write permissions for the new directory.

Step 15: Restart the first config server with MMAPv1. Specify the newly created MMAPv1 data directory as the `--dbpath` as well as any additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Step 16: Upload the exported data using `mongorestore` to the first config server.

```
mongorestore <exportDataDestination>
```

Specify additional options as appropriate. See `mongorestore` for available options.

Step 17: Enable writes to the sharded cluster’s metadata. Restart the second config server, specifying the newly created MMAPv1 data directory as the `--dbpath`. Specify additional options as appropriate.

```
mongod --dbpath <newMMAPv1DBPath> --configsvr
```

Once all three config servers are up, the sharded cluster’s metadata is available for writes.

Step 18: Re-enable the balancer. Once all three config servers are up and running with WiredTiger, `Re-enable the balancer` (page 718).
Downgrade Binaries  Once upgraded to MongoDB 3.0, you cannot downgrade to a version lower than 2.6.5.

The downgrade procedure for a sharded cluster reverses the order of the upgrade procedure. The version v6 config database is backwards compatible with MongoDB 2.6.

Step 1: Disable the Balancer.  Turn off the balancer (page 684) in the sharded cluster, as described in Disable the Balancer (page 717).

Step 2: Downgrade each shard, one at a time.  For each shard:

1. Downgrade the mongod secondaries before downgrading the primary.
2. To downgrade the primary, run replSetStepDown and downgrade.

Step 3: Downgrade the config servers.  Downgrade all 3 mongod config server instances, leaving the first system in the mongos --configdb argument to downgrade last.

Step 4: Downgrade the mongos instances.  Downgrade and restart each mongos, one at a time. The downgrade process is a binary drop-in replacement.

Step 5: Re-enable the balancer.  Once the upgrade of sharded cluster components is complete, re-enable the balancer (page 718).

General Downgrade Procedure  Except as described on this page, moving between 2.6 and 3.0 is a drop-in replacement:

Step 1: Stop the existing mongod instance.  For example, on Linux, run 3.0 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 225) 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.

See Upgrade MongoDB to 3.0 (page 810) for full upgrade instructions.

Download

To download MongoDB 3.0, go to the downloads page282.

See also:

- All Third Party License Notices283.

---

282 http://www.mongodb.org/downloads
283 https://github.com/mongodb/mongo/blob/v3.0/distsrc/THIRD-PARTY-NOTICES
• All JIRA issues resolved in 3.0\textsuperscript{284}.

**Additional Resources**

• Blog Post: Announcing MongoDB 3.0\textsuperscript{285}
• Whitepaper: What’s New in MongoDB 3.0\textsuperscript{286}
• Webinar: What’s New in MongoDB 3.0\textsuperscript{287}

### 12.2 Previous Stable Releases

#### 12.2.1 Release Notes for MongoDB 2.6

*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

**2.6.10 – Changes**

**Security**

• SERVER-18312\textsuperscript{288} Upgrade PCRE to latest
• SERVER-17812\textsuperscript{289} LockPinger has audit-related GLE failure
• SERVER-17647\textsuperscript{290} Compute BinData length in v8
• SERVER-17591\textsuperscript{291} Add SSL flag to select supported protocols
• SERVER-16849\textsuperscript{292} On mongos we always invalidate the user cache once, even if no user definitions are changing
• SERVER-11980\textsuperscript{293} Improve user cache invalidation enforcement on mongos

**Querying**

• SERVER-18364\textsuperscript{294} Ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction

\textsuperscript{284}http://bit.ly/1CpOu6t
\textsuperscript{285}http://www.mongodb.com/blog/post/announcing-mongodb-3-0?jmp=docs
\textsuperscript{286}https://www.mongodb.com/lp/white-paper/mongodb-3-0?jmp=docs
\textsuperscript{287}https://www.mongodb.com/webinar/Whats-New-in-MongoDB-3-0?jmp=docs
\textsuperscript{288}https://jira.mongodb.org/browse/SERVER-18312
\textsuperscript{289}https://jira.mongodb.org/browse/SERVER-17812
\textsuperscript{290}https://jira.mongodb.org/browse/SERVER-17647
\textsuperscript{291}https://jira.mongodb.org/browse/SERVER-17591
\textsuperscript{292}https://jira.mongodb.org/browse/SERVER-16849
\textsuperscript{293}https://jira.mongodb.org/browse/SERVER-11980
\textsuperscript{294}https://jira.mongodb.org/browse/SERVER-18364
• SERVER-17815 Plan ranking tie breaker is computed incorrectly
• SERVER-16256 $all clause with elemMatch uses wider bounds than needed

**Replication**
• SERVER-18211 MongoDB fails to correctly roll back collection creation
• SERVER-17771 Reconfiguring a replica set to remove a node causes a segmentation fault on 2.6.8
• SERVER-13542 Expose electionId on primary in isMaster

**Sharding**
• SERVER-17812 LockPinger has audit-related GLE failure
• SERVER-17805 logOp / OperationObserver should always check shardversion
• SERVER-17749 collMod usePowerOf2Sizes fails on mongos
• SERVER-11980 Improve user cache invalidation enforcement on mongos

**Storage**
• SERVER-18211 MongoDB fails to correctly roll back collection creation
• SERVER-17653 ERROR: socket XXX is higher than 1023; not supported on 2.6.*

**Indexing**
• SERVER-17018 Assertion failure false src/mongo/db/structure/btree/key.cpp Line 433 on remove operation

**Write Ops**
• SERVER-18111 mongod allows user inserts into system.profile collection
• SERVER-13542 Expose electionId on primary in isMaster

**Networking**
• SERVER-18096 Shard primary incorrectly reuses closed sockets after relinquish and re-election
• SERVER-17591 Add SSL flag to select supported protocols
Build and Packaging

- SERVER-18344 logs should be sent to updated logkeeper server
- SERVER-18082 Change smoke.py buildlogger command line options to environment variables
- SERVER-18312 Upgrade PCRE to latest
- SERVER-17780 Init script sets process ulimit to different value compared to documentation
- SERVER-16563 Debian repo component mismatch - mongodb/10gen

Shell SERVER-17951 db.currentOp() fails with read preference set

Testing

- SERVER-18262 setup_multiversion_mongodb should retry links download on timeouts
- SERVER-18229 smoke.py with PyMongo 3.0.1 fails to run certain tests
- SERVER-18073 Fix smoke.py to work with PyMongo 3.0

2.6.9 – Changes

Security SERVER-16073 Create hidden net.ssl.sslCipherConfig flag

Querying

- SERVER-14723 Crash during query planning for geoNear with multiple 2dsphere indexes
- SERVER-14071 For queries with sort(), bad non-blocking plan can be cached if there are zero results
- SERVER-8188 Configurable idle cursor timeout

Replication and Sharding

- SERVER-17429 the message logged when changing sync target due to stale data should format OpTimes in a consistent way
- SERVER-17441 mongos crash right after “not master” error

Storage SERVER-15907 Use ftruncate rather than fallocate when running on tmpfs

311 https://jira.mongodb.org/browse/SERVER-18344
312 https://jira.mongodb.org/browse/SERVER-18082
313 https://jira.mongodb.org/browse/SERVER-18312
314 https://jira.mongodb.org/browse/SERVER-17780
315 https://jira.mongodb.org/browse/SERVER-16563
316 https://jira.mongodb.org/browse/SERVER-17951
317 https://jira.mongodb.org/browse/SERVER-18262
318 https://jira.mongodb.org/browse/SERVER-18229
319 https://jira.mongodb.org/browse/SERVER-18073
320 https://jira.mongodb.org/browse/SERVER-16073
321 https://jira.mongodb.org/browse/SERVER-14723
322 https://jira.mongodb.org/browse/SERVER-14071
323 https://jira.mongodb.org/browse/SERVER-8188
324 https://jira.mongodb.org/browse/SERVER-17429
325 https://jira.mongodb.org/browse/SERVER-17441
326 https://jira.mongodb.org/browse/SERVER-15907
Aggregation Framework

• SERVER-17426327 Aggregation framework query by _id returns duplicates in sharded cluster (orphan documents)
• SERVER-17224328 Aggregation pipeline with 64MB document can terminate server

Build and Platform

• SERVER-17484329 Migrate server MCI config into server repo
• SERVER-17252330 Upgrade PCRE Version from 8.30 to Latest

Diagnostics and Internal Code

• SERVER-17226331 top command with 64MB result document can terminate server
• SERVER-17338332 NULL pointer crash when running copydb against stepped-down 2.6 primary
• SERVER-14992333 Query for Windows 7 File Allocation Fix, and other hotfixes

2.6.8 – Changes

Security and Networking

• SERVER-17278334 BSON BinData validation enforcement
• SERVER-17022335 No SSL Session Caching may not be respected
• SERVER-17264336 improve bson validation

Query and Aggregation

• SERVER-16655337 Geo predicate is unable to use compound 2dsphere index if it is root of $or clause
• SERVER-16527338 2dsphere explain reports “works” for nscanned & nscannedObjects
• SERVER-15802339 Query optimizer should always use equality predicate over unique index when possible
• SERVER-14044340 Incorrect ${meta: ‘text’} reference in aggregation $sort error message

---

327 https://jira.mongodb.org/browse/SERVER-17426
328 https://jira.mongodb.org/browse/SERVER-17224
329 https://jira.mongodb.org/browse/SERVER-17484
330 https://jira.mongodb.org/browse/SERVER-17252
331 https://jira.mongodb.org/browse/SERVER-17226
332 https://jira.mongodb.org/browse/SERVER-17338
333 https://jira.mongodb.org/browse/SERVER-14992
334 https://jira.mongodb.org/browse/SERVER-17278
335 https://jira.mongodb.org/browse/SERVER-17022
336 https://jira.mongodb.org/browse/SERVER-17264
337 https://jira.mongodb.org/browse/SERVER-16655
338 https://jira.mongodb.org/browse/SERVER-16527
339 https://jira.mongodb.org/browse/SERVER-15802
340 https://jira.mongodb.org/browse/SERVER-14044
Replication

- SERVER-16599\(^{341}\) copydb and clone commands can crash the server if a primary steps down
- SERVER-16315\(^{342}\) Replica set nodes should not threaten to veto nodes whose config version is higher than their own
- SERVER-16274\(^{343}\) secondary fasserts trying to replicate an index
- SERVER-15471\(^{344}\) Better error message when replica is not found in GhostSync::associateSlave

Sharding

- SERVER-17191\(^{345}\) Spurious warning during upgrade of sharded cluster
- SERVER-17163\(^{346}\) Fatal error “logOp but not primary” in MigrateStatus::go
- SERVER-16984\(^{347}\) UpdateLifecycleImpl can return empty collectionMetadata even if ns is sharded
- SERVER-10904\(^{348}\) Possible for _master and _slaveConn to be pointing to different connections even with primary read pref

Storage

- SERVER-17087\(^{349}\) Add listCollections command functionality to 2.6 shell & client
- SERVER-14572\(^{350}\) Increase C runtime stdio file limit

Tools

- SERVER-17216\(^{351}\) 2.6 mongostat cannot be used with 3.0 mongod
- SERVER-14190\(^{352}\) mongorestore parseMetadataFile passes non-null terminated string to 'fromjson'

Build and Packaging

- SERVER-14803\(^{353}\) Support static libstdc++ builds for non-Linux builds
- SERVER-15400\(^{354}\) Create Windows Enterprise Zip File with vcredist and dependent dlls

Usability

SERVER-14756\(^{355}\) The YAML storage.quota.enforced option is not found

---

\(^{341}\) https://jira.mongodb.org/browse/SERVER-16599
\(^{342}\) https://jira.mongodb.org/browse/SERVER-16315
\(^{343}\) https://jira.mongodb.org/browse/SERVER-16274
\(^{344}\) https://jira.mongodb.org/browse/SERVER-15471
\(^{345}\) https://jira.mongodb.org/browse/SERVER-17191
\(^{346}\) https://jira.mongodb.org/browse/SERVER-17163
\(^{347}\) https://jira.mongodb.org/browse/SERVER-16984
\(^{348}\) https://jira.mongodb.org/browse/SERVER-10904
\(^{349}\) https://jira.mongodb.org/browse/SERVER-17087
\(^{350}\) https://jira.mongodb.org/browse/SERVER-14572
\(^{351}\) https://jira.mongodb.org/browse/SERVER-17216
\(^{352}\) https://jira.mongodb.org/browse/SERVER-14190
\(^{353}\) https://jira.mongodb.org/browse/SERVER-14803
\(^{354}\) https://jira.mongodb.org/browse/SERVER-15400
\(^{355}\) https://jira.mongodb.org/browse/SERVER-14756
Testing  SERVER-16421\textsuperscript{356} sharding_rs2.js should clean up data on all replicas

2.6.7 – Changes

Stability
- SERVER-16237\textsuperscript{357} Don’t check the shard version if the primary server is down

Querying
- SERVER-16408\textsuperscript{358} max_time_ms.js should not run in parallel suite.

Replication
- SERVER-16732\textsuperscript{359} SyncSourceFeedback::replHandshake() may perform an illegal erase from a std::map in some circumstances

Sharding
- SERVER-16683\textsuperscript{360} Decrease mongos memory footprint when shards have several tags
- SERVER-15766\textsuperscript{361} prefix_shard_key.js depends on primary allocation to particular shards
- SERVER-14306\textsuperscript{362} mongos can cause shards to hit the in-memory sort limit by requesting more results than needed.

Packaging
- SERVER-16081\textsuperscript{363} /etc/init.d/mongod startup script fails, with dirname message

2.6.6 – Changes

Security
- SERVER-15673\textsuperscript{364} Disable SSLv3 ciphers
- SERVER-15515\textsuperscript{365} New test for mixed version replSet, 2.4 primary, user updates
- SERVER-15500\textsuperscript{366} New test for system.user operations
Stability

- SERVER-12061\(^{367}\) Do not silently ignore read errors when syncing a replica set node
- SERVER-12058\(^{368}\) Primary should abort if encountered problems writing to the oplog

Querying

- SERVER-16291\(^{369}\) Cannot set/list/clear index filters on the secondary
- SERVER-15958\(^{370}\) The “isMultiKey” value is not correct in the output of aggregation explain plan
- SERVER-15899\(^{371}\) Querying against path in document containing long array of subdocuments with nested arrays causes stack overflow
- SERVER-15696\(^{372}\) $\text{regex}$, $\text{in}$ and $\text{sort}$ with index returns too many results
- SERVER-15639\(^{373}\) Text queries can return incorrect results and leak memory when multiple predicates given on same text index prefix field
- SERVER-15580\(^{374}\) Evaluating candidate query plans with concurrent writes on same collection may crash mongod
- SERVER-15528\(^{375}\) Distinct queries can scan many index keys without yielding read lock
- SERVER-15485\(^{376}\) CanonicalQuery::canonicalize can leak a LiteParsedQuery
- SERVER-15403\(^{377}\) $\text{min}$ and $\text{max}$ equal errors in 2.6 but not in 2.4
- SERVER-15233\(^{378}\) Cannot run planCacheListQueryShapes on a Secondary
- SERVER-14799\(^{379}\) $\text{count}$ with hint doesn’t work when hint is a document

Replication

- SERVER-16107\(^{380}\) 2.6 mongod crashes with segfault when added to a 2.8 replica set with $\geq 12$ nodes.
- SERVER-15994\(^{381}\) listIndexes and listCollections can be run on secondaries without slaveOk bit
- SERVER-15849\(^{382}\) do not forward replication progress for nodes that are no longer part of a replica set
- SERVER-15491\(^{383}\) SyncSourceFeedback can crash due to a SocketException in authenticateInternalUser

\(^{367}\)https://jira.mongodb.org/browse/SERVER-12061
\(^{368}\)https://jira.mongodb.org/browse/SERVER-12058
\(^{369}\)https://jira.mongodb.org/browse/SERVER-16291
\(^{370}\)https://jira.mongodb.org/browse/SERVER-15958
\(^{371}\)https://jira.mongodb.org/browse/SERVER-15899
\(^{372}\)https://jira.mongodb.org/browse/SERVER-15696
\(^{373}\)https://jira.mongodb.org/browse/SERVER-15639
\(^{374}\)https://jira.mongodb.org/browse/SERVER-15580
\(^{375}\)https://jira.mongodb.org/browse/SERVER-15528
\(^{376}\)https://jira.mongodb.org/browse/SERVER-15485
\(^{377}\)https://jira.mongodb.org/browse/SERVER-15403
\(^{378}\)https://jira.mongodb.org/browse/SERVER-15233
\(^{379}\)https://jira.mongodb.org/browse/SERVER-14799
\(^{380}\)https://jira.mongodb.org/browse/SERVER-16107
\(^{381}\)https://jira.mongodb.org/browse/SERVER-15994
\(^{382}\)https://jira.mongodb.org/browse/SERVER-15849
\(^{383}\)https://jira.mongodb.org/browse/SERVER-15491
Sharding

- SERVER-15318384 copydb should not use exhaust flag when used against mongos
- SERVER-14728385 Shard depends on string comparison of replica set connection string
- SERVER-14506386 special top chunk logic can move max chunk to a shard with incompatible tag
- SERVER-14299387 For sharded limit=N queries with sort, mongos can request >N results from shard
- SERVER-14080388 Have migration result reported in the changelog correctly
- SERVER-12472389 Fail MoveChunk if an index is needed on TO shard and data exists

Storage

- SERVER-16283390 Can’t start new wiredtiger node with log file or config file in data directory - false detection of old mmapv1 files
- SERVER-15986391 Starting with different storage engines in the same dbpath should error/warn
- SERVER-14057392 Changing TTL expiration time with collMod does not correctly update index definition

Indexing and write Operations

- SERVER-14287393 ensureIndex can abort reIndex and lose indexes
- SERVER-14886394 Updates against paths composed with array index notation and positional operator fail with error

Data Aggregation

Data Aggregation SERVER-15552395 Errors writing to temporary collections during mapReduce command execution should be operation-fatal

Build and Packaging

- SERVER-14184396 Unused preprocessor macros from s2 conflict on OS X Yosemite
- SERVER-14015397 S2 Compilation on GCC 4.9/Solaris fails
- SERVER-16017398 Suse11 enterprise packages fail due to unmet dependencies
- SERVER-15598399 Ubuntu 14.04 Enterprise packages depend on unavailable libsnmp15 package
- SERVER-13595400 Red Hat init.d script error: YAML config file parsing

384 https://jira.mongodb.org/browse/SERVER-15318
385 https://jira.mongodb.org/browse/SERVER-14728
386 https://jira.mongodb.org/browse/SERVER-14506
387 https://jira.mongodb.org/browse/SERVER-14299
388 https://jira.mongodb.org/browse/SERVER-14080
389 https://jira.mongodb.org/browse/SERVER-12472
390 https://jira.mongodb.org/browse/SERVER-16283
391 https://jira.mongodb.org/browse/SERVER-15986
392 https://jira.mongodb.org/browse/SERVER-14057
393 https://jira.mongodb.org/browse/SERVER-14287
394 https://jira.mongodb.org/browse/SERVER-14886
395 https://jira.mongodb.org/browse/SERVER-15552
396 https://jira.mongodb.org/browse/SERVER-14184
397 https://jira.mongodb.org/browse/SERVER-14015
398 https://jira.mongodb.org/browse/SERVER-16017
399 https://jira.mongodb.org/browse/SERVER-15598
400 https://jira.mongodb.org/browse/SERVER-13595
Logging and Diagnostics

- SERVER-13471 Increase log level of “did reduceInMemory” message in map/reduce
- SERVER-16324 Command execution log line displays “query not recording (too large)” instead of abbreviated command object
- SERVER-10069 Improve errorcodes.py so it captures multiline messages

Testing and Internals

- SERVER-15632 MultiHostQueryOp::PendingQueryContext::doBlockingQuery can leak a cursor object
- SERVER-15629 GeoParser::parseMulti{Line|Polygon} does not clear objects owned by out parameter
- SERVER-16316 Remove unsupported behavior in shard3.js
- SERVER-14763 Update jstests/sharding/split_large_key.js
- SERVER-14249 Add tests for querying oplog via mongodump using –dbpath
- SERVER-13726 indexbg_drop.js

2.6.5 – Changes

Security

- SERVER-15465 OpenSSL crashes on stepdown
- SERVER-15360 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 Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14727 Details of SASL failures aren’t logged
- SERVER-12551 Audit DML/CRUD operations

Stability

SERVER-9032 mongod fails when launched with misconfigured locale

401 https://jira.mongodb.org/browse/SERVER-13471
402 https://jira.mongodb.org/browse/SERVER-16324
403 https://jira.mongodb.org/browse/SERVER-10069
404 https://jira.mongodb.org/browse/SERVER-15632
405 https://jira.mongodb.org/browse/SERVER-15629
406 https://jira.mongodb.org/browse/SERVER-16316
407 https://jira.mongodb.org/browse/SERVER-14763
408 https://jira.mongodb.org/browse/SERVER-14249
409 https://jira.mongodb.org/browse/SERVER-13726
410 https://jira.mongodb.org/browse/SERVER-15465
411 https://jira.mongodb.org/browse/SERVER-15360
412 https://jira.mongodb.org/browse/SERVER-14887
413 https://jira.mongodb.org/browse/SERVER-14727
414 https://jira.mongodb.org/browse/SERVER-12551
415 https://jira.mongodb.org/browse/SERVER-9032
Querying

- SERVER-15287 Query planner sort analysis incorrectly allows index key pattern plugin fields to provide sort
- SERVER-15286 Assertion in date indexes when opposite-direction-sorted and double “or” filtered
- SERVER-15279 Disable hash-based index intersection (AND_HASH) by default
- SERVER-15152 When evaluating plans, some index candidates cause complete index scan
- SERVER-15015 Assertion failure when combining $max and $min and reverse index scan
- SERVER-15012 Server crashes on indexed rooted $or queries using a 2d index
- SERVER-14969 Dropping index during active aggregation operation can crash server
- SERVER-14961 Plan ranker favors intersection plans if predicate generates empty range index scan
- SERVER-14892 Invalid {$elemMatch: {$where}) query causes memory leak
- SERVER-14706 Queries that use negated $type predicate over a field may return incomplete results when an index is present on that field
- SERVER-13104 Plan enumerator doesn’t enumerate all possibilities for a nested $or
- SERVER-14984 Server aborts when running $centerSphere query with NaN radius
- SERVER-14981 Server aborts when querying against 2dsphere index with coarsestIndexedLevel:0
- SERVER-14831 Text search trips assertion when default language only supported in textIndexVersion=1 used

Replication

- SERVER-15038 Multiple background index builds may not interrupt cleanly for commands, on secondaries
- SERVER-14887 Allow user document changes made on a 2.4 primary to replicate to a 2.6 secondary
- SERVER-14805 Use multithreaded oplog replay during initial sync

Sharding

- SERVER-15056 Sharded connection cleanup on setup error can crash mongos
- SERVER-13702 Commands without optional query may target to wrong shards on mongos
• SERVER-15156 MongoDB upgrade 2.4 to 2.6 check returns error in `config.changelog` collection

Storage
• SERVER-15369 explicitly zero .ns files on creation
• SERVER-15319 Verify 2.8 freelist is upgrade-downgrade safe with 2.6
• SERVER-15111 partially written journal last section causes recovery to fail

Indexing
• SERVER-14848 Port `index_id_desc.js` to v2.6 and master branches
• SERVER-14205 `ensureIndex` failure reports `ok: 1` on some failures

Write Operations
• SERVER-15106 Incorrect nscanned and nscannedObjects for idhack updates in 2.6.4 profiler or slow query log
• SERVER-15029 The `$rename` modifier uses incorrect dotted source path
• SERVER-14829 `UpdateIndexData::clear()` should reset all member variables

Data Aggregation
• SERVER-15087 Server crashes when running concurrent mapReduce and dropDatabase commands
• SERVER-14969 Dropping index during active aggregation operation can crash server
• SERVER-14168 Warning logged when incremental MR collections are unsuccessfully dropped on secondaries

Packaging
• SERVER-14679 (CentOS 7/RHEL 7) `init.d` script should create directory for `pid` file if it is missing
• SERVER-14023 Support for RHEL 7 Enterprise `.rpm` packages
• SERVER-13243 Support for Ubuntu 14 “Trusty” Enterprise `.deb` packages
• SERVER-11077 Support for Debian 7 Enterprise `.deb` packages
• SERVER-10642 Generate Community and Enterprise packages for SUSE 11

12.2. Previous Stable Releases
Logging and Diagnostics

- SERVER-14964: nsCanned not written to the logs at logLevel 1 unless slowms exceeded or profiling enabled
- SERVER-12551: Audit DML/CRUD operations
- SERVER-14904: Adjust dates in tool/exportimport_date.js to account for different timezones

Internal Code and Testing

- SERVER-13770: Helpers::removeRange should check all runner states
- SERVER-14284: jstests should not leave profiler enabled at test run end
- SERVER-14076: remove test replset_remove_node.js
- SERVER-14778: Hide function and data pointers for natively-injected v8 functions

2.6.4 – Changes

Security

- SERVER-14701: The “backup” auth role should allow running the “collstats” command for all resources
- SERVER-14518: Allow disabling hostname validation for SSL
- SERVER-14268: Potential information leak
- SERVER-14170: Cannot read from secondary if both audit and auth are enabled in a sharded cluster
- SERVER-13833: userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles
- SERVER-12512: Add role-based, selective audit logging.
- SERVER-9482: Add build flag for sslFIPSMo

Querying

- SERVER-14625: Query planner can construct incorrect bounds for negations inside $elemMatch
- SERVER-14607: hash intersection of fetched and non-fetched data can discard data from a result
- SERVER-14532: Improve logging in the case of plan ranker ties

452 https://jira.mongodb.org/browse/SERVER-14964
453 https://jira.mongodb.org/browse/SERVER-12551
454 https://jira.mongodb.org/browse/SERVER-14904
455 https://jira.mongodb.org/browse/SERVER-13770
456 https://jira.mongodb.org/browse/SERVER-14284
457 https://jira.mongodb.org/browse/SERVER-14076
458 https://jira.mongodb.org/browse/SERVER-14778
459 https://jira.mongodb.org/browse/SERVER-14701
460 https://jira.mongodb.org/browse/SERVER-14518
461 https://jira.mongodb.org/browse/SERVER-14268
462 https://jira.mongodb.org/browse/SERVER-14170
463 https://jira.mongodb.org/browse/SERVER-13833
464 https://jira.mongodb.org/browse/SERVER-12512
465 https://jira.mongodb.org/browse/SERVER-9482
466 https://jira.mongodb.org/browse/SERVER-14625
467 https://jira.mongodb.org/browse/SERVER-14607
468 https://jira.mongodb.org/browse/SERVER-14532
• SERVER-14350469 Server crash when $centerSphere has non-positive radius
• SERVER-14317470 Dead code in IDHackRunner::applyProjection
• SERVER-14311471 skipping of index keys is not accounted for in plan ranking by the index scan stage
• SERVER-14123472 some operations can create BSON object larger than the 16MB limit
• SERVER-14034473 Sorted $in query with large number of elements can’t use merge sort
• SERVER-13994474 do not aggressively pre-fetch data for parallelCollectionScan

Replication
• SERVER-14665475 Build failure for v2.6 in closeall.js caused by access violation reading _me
• SERVER-14505476 cannot dropAllIndexes when index builds in progress assertion failure
• SERVER-14494477 Dropping collection during active background index build on secondary triggers segfault
• SERVER-13822478 Running resync before replset config is loaded can crash mongod
• SERVER-11776479 Replication ‘isself’ check should allow mapped ports

Sharding
• SERVER-14551480 Runner yield during migration cleanup (removeRange) results in fassert
• SERVER-14431481 Invalid chunk data after splitting on a key that’s too large
• SERVER-14261482 stepdown during migration range delete can abort mongod
• SERVER-14032483 v2.6 mongos doesn’t verify _id is present for config server upserts
• SERVER-13648484 better stats from migration cleanup
• SERVER-12750485 mongos shouldn’t accept initial query with “exhaust” flag set
• SERVER-9788486 mongos does not re-evaluate read preference once a valid replica set member is chosen
• SERVER-9526487 Log messages regarding chunks not very informative when the shard key is of type BinData

469https://jira.mongodb.org/browse/SERVER-14350
470https://jira.mongodb.org/browse/SERVER-14317
471https://jira.mongodb.org/browse/SERVER-14311
472https://jira.mongodb.org/browse/SERVER-14123
473https://jira.mongodb.org/browse/SERVER-14034
474https://jira.mongodb.org/browse/SERVER-13994
475https://jira.mongodb.org/browse/SERVER-14665
476https://jira.mongodb.org/browse/SERVER-14505
477https://jira.mongodb.org/browse/SERVER-14494
478https://jira.mongodb.org/browse/SERVER-13822
479https://jira.mongodb.org/browse/SERVER-11776
480https://jira.mongodb.org/browse/SERVER-14551
481https://jira.mongodb.org/browse/SERVER-14431
482https://jira.mongodb.org/browse/SERVER-14261
483https://jira.mongodb.org/browse/SERVER-14032
484https://jira.mongodb.org/browse/SERVER-13648
485https://jira.mongodb.org/browse/SERVER-12750
486https://jira.mongodb.org/browse/SERVER-9788
487https://jira.mongodb.org/browse/SERVER-9526
MongoDB Documentation, Release 3.0.4

Storage

• SERVER-14198 Std::set<pointer> and Windows Heap Allocation Reuse produces non-deterministic results
• SERVER-13975 Creating index on collection named “system” can cause server to abort
• SERVER-13729 Reads & Writes are blocked during data file allocation on Windows
• SERVER-13681 mongod B stalls during background flush on Windows

Indexing  SERVER-14494 Dropping collection during active background index build on secondary triggers seg-fault

Write Ops

• SERVER-14257 “remove” command can cause process termination by throwing unhandled exception if profiling is enabled
• SERVER-14024 Update fails when query contains part of a DBRef and results in an insert (upsert:true)
• SERVER-13764 debug mechanisms report incorrect nscanned / nscannedObjects for updates

Networking  SERVER-13734 Remove catch (...) from handleIncomingMsg

Geo

• SERVER-14039 $nearSphere query with 2d index, skip, and limit returns incomplete results
• SERVER-13701 Query using 2d index throws exception when using explain()

Text Search

• SERVER-14738 Updates to documents with text-indexed fields may lead to incorrect entries
• SERVER-14027 Renaming collection within same database fails if wildcard text index present

Tools

• SERVER-14212 mongorestore may drop system users and roles
• SERVER-14048 mongodump against mongos can’t send dump to standard output

488 https://jira.mongodb.org/browse/SERVER-14198
489 https://jira.mongodb.org/browse/SERVER-13975
490 https://jira.mongodb.org/browse/SERVER-13729
491 https://jira.mongodb.org/browse/SERVER-13681
492 https://jira.mongodb.org/browse/SERVER-14494
493 https://jira.mongodb.org/browse/SERVER-14257
494 https://jira.mongodb.org/browse/SERVER-14024
495 https://jira.mongodb.org/browse/SERVER-13764
496 https://jira.mongodb.org/browse/SERVER-13701
497 https://jira.mongodb.org/browse/SERVER-14039
498 https://jira.mongodb.org/browse/SERVER-14738
499 https://jira.mongodb.org/browse/SERVER-14027
500 https://jira.mongodb.org/browse/SERVER-14212
501 https://jira.mongodb.org/browse/SERVER-14048

Chapter 12. Release Notes
Admin

- SERVER-14556 Default dbpath for `mongod --configsvr` changes in 2.6
- SERVER-14355 Allow dbAdmin role to manually create system.profile collections

Packaging  SERVER-14283 Parameters in installed config file are out of date

JavaScript

- SERVER-14254 Do not store native function pointer as a property in function prototype
- SERVER-13798 v8 garbage collection can cause crash due to independent lifetime of DBClient and Cursor objects
- SERVER-13707 mongo shell may crash when converting invalid regular expression

Shell

- SERVER-14341 negative opcounter values in serverStatus
- SERVER-14107 Querying for a document containing a value of either type Javascript or JavascriptWithScope crashes the shell

Usability  SERVER-13833 userAdminAnyDatabase role should be able to create indexes on admin.system.users and admin.system.roles

Logging and Diagnostics

- SERVER-12512 Add role-based, selective audit logging.
- SERVER-14341 negative opcounter values in serverStatus

Testing

- SERVER-14731 plan_cache_ties.js sometimes fails
- SERVER-14147 make index_multi.js retry on connection failure
- SERVER-13615 sharding_rs2.js intermittent failure due to reliance on opcounters

503: https://jira.mongodb.org/browse/SERVER-14556
504: https://jira.mongodb.org/browse/SERVER-14355
505: https://jira.mongodb.org/browse/SERVER-14283
506: https://jira.mongodb.org/browse/SERVER-14254
507: https://jira.mongodb.org/browse/SERVER-13798
508: https://jira.mongodb.org/browse/SERVER-13707
509: https://jira.mongodb.org/browse/SERVER-14341
510: https://jira.mongodb.org/browse/SERVER-14107
511: https://jira.mongodb.org/browse/SERVER-13833
512: https://jira.mongodb.org/browse/SERVER-12512
513: https://jira.mongodb.org/browse/SERVER-14341
514: https://jira.mongodb.org/browse/SERVER-14731
515: https://jira.mongodb.org/browse/SERVER-14147
516: https://jira.mongodb.org/browse/SERVER-13615
2.6.3 – Changes

- SERVER-14302: Fixed: “Equality queries on _id with projection may return no results on sharded collections”
- SERVER-14304: Fixed: “Equality queries on _id with projection on _id may return orphan documents on sharded collections”

2.6.2 – Changes

Security

- SERVER-13727: The backup (page 410) authorization role now includes privileges to run the collStats command.
- SERVER-13804: The built-in role restore (page 411) now has privileges on system.roles collection.
- SERVER-13612: Fixed: “SSL-enabled server appears not to be sending the list of supported certificate issuers to the client”
- SERVER-13753: Fixed: “mongod may terminate if x.509 authentication certificate is invalid”
- SERVER-13945: For replica set/sharded cluster member authentication (page 363), now matches x.509 cluster certificates by attributes instead of by substring comparison.
- SERVER-13868: Now marks V1 users as probed on databases that do not have surrogate user documents.
- SERVER-13850: 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: Fixed: “Shell prints startup warning when auth enabled”

Querying

- SERVER-13731: Fixed: “Stack overflow when parsing deeply nested $not query”
- SERVER-13890: Fixed: “Index bounds builder constructs invalid bounds for multiple negations joined by an $or”
- SERVER-13752: Verified assertion on empty $in clause and sort on second field in a compound index.
- SERVER-13337: Re-enabled idhack for queries with projection.
- SERVER-13715: Fixed: “Aggregation pipeline execution can fail with $or and blocking sorts”
- SERVER-13714: Fixed: “non-top-level indexable $not triggers query planning bug”

517 https://jira.mongodb.org/browse/SERVER-14302
518 https://jira.mongodb.org/browse/SERVER-14304
519 https://jira.mongodb.org/browse/SERVER-13727
520 https://jira.mongodb.org/browse/SERVER-13804
521 https://jira.mongodb.org/browse/SERVER-13612
522 https://jira.mongodb.org/browse/SERVER-13753
523 https://jira.mongodb.org/browse/SERVER-13945
524 https://jira.mongodb.org/browse/SERVER-13868
525 https://jira.mongodb.org/browse/SERVER-13850
526 https://jira.mongodb.org/browse/SERVER-13588
527 https://jira.mongodb.org/browse/SERVER-13731
528 https://jira.mongodb.org/browse/SERVER-13890
529 https://jira.mongodb.org/browse/SERVER-13752
530 https://jira.mongodb.org/browse/SERVER-13337
531 https://jira.mongodb.org/browse/SERVER-13715
532 https://jira.mongodb.org/browse/SERVER-13714
- SERVER-13769 Fixed: “distinct command on indexed field with geo predicate fails to execute”
- SERVER-13675 Fixed “Plans with differing performance can tie during plan ranking”
- SERVER-13899 Fixed: “Whole index scan’ query solutions can use incompatible indexes, return incorrect results”
- SERVER-13852 Fixed “IndexBounds::endKeyInclusive not initialized by constructor”
- SERVER-14073 planSummary no longer truncated at 255 characters
- SERVER-14174 Fixed: “If ntoreturn is a limit (rather than batch size) extra data gets buffered during plan ranking”
- SERVER-13789 Some nested queries no longer trigger an assertion error
- SERVER-14064 Added planSummary information for count command log message.
- SERVER-13960 Queries containing $or no longer miss results if multiple clauses use the same index.
- SERVER-14180 Fixed: “Crash with ‘and’ clause, $elemMatch, and nested $mod or regex”
- SERVER-14176 Natural order sort specification no longer ignored if query is specified.
- SERVER-13754 Bounds no longer combined for $or queries that can use merge sort.

**Geospatial**  SERVER-13687 Results of $near query on compound multi-key 2dsphere index are now sorted by distance.

**Write Operations**  SERVER-13802 Insert field validation no longer stops at first Timestamp() field.

**Replication**
- SERVER-13993 Fixed: “log a message when shouldChangeSyncTarget() believes a node should change sync targets”
- SERVER-13976 Fixed: “Cloner needs to detect failure to create collection”

**Sharding**
- SERVER-13616 Resolved: “‘type 7’ (OID) error when acquiring distributed lock for first time”
- SERVER-13812 Now catches exception thrown by getShardsForQuery for geo query.

---

533https://jira.mongodb.org/browse/SERVER-13769
534https://jira.mongodb.org/browse/SERVER-13675
535https://jira.mongodb.org/browse/SERVER-13899
536https://jira.mongodb.org/browse/SERVER-13852
537https://jira.mongodb.org/browse/SERVER-14073
538https://jira.mongodb.org/browse/SERVER-14174
539https://jira.mongodb.org/browse/SERVER-13754
540https://jira.mongodb.org/browse/SERVER-14064
541https://jira.mongodb.org/browse/SERVER-13960
542https://jira.mongodb.org/browse/SERVER-14180
543https://jira.mongodb.org/browse/SERVER-14176
544https://jira.mongodb.org/browse/SERVER-13812
545https://jira.mongodb.org/browse/SERVER-13687
546https://jira.mongodb.org/browse/SERVER-13802
547https://jira.mongodb.org/browse/SERVER-13993
548https://jira.mongodb.org/browse/SERVER-13976
549https://jira.mongodb.org/browse/SERVER-13616
550https://jira.mongodb.org/browse/SERVER-13812
• SERVER-14138 mongos will now correctly target multiple shards for nested field shard key predicates.
• SERVER-11332 Fixed: “Authentication requests delayed if first config server is unresponsive”

Map/Reduce
• SERVER-14186 Resolved: “rs.stepDown during mapReduce causes fassert in logOp”
• SERVER-13981 Temporary mapreduce collections are now correctly replicated to secondaries.

Storage
• SERVER-13750 convertToCapped on empty collection no longer aborts after invariant() failure.
• SERVER-14056 Moving large collection across databases with renameCollection no longer triggers fatal assertion.
• SERVER-14082 Fixed: “Excessive freelist scanning for MaxBucket”
• SERVER-13737 CollectionOptions parser now skips non-numeric for “size”/”max” elements if values non-numeric.

Build and Packaging
• SERVER-13950 MongoDB Enterprise now includes required dependency list.
• SERVER-13862 Support for mongodb-org-server installation 2.6.1-1 on RHEL5 via RPM.
• SERVER-13724 Added SCons flag to override treating all warnings as errors.

Diagnostics
• SERVER-13587 Resolved: “ndeleted (page 294) in system.profile documents reports 1 too few documents removed”
• SERVER-13368 Improved exposure of timing information in currentOp.

Administration
• SERVER-13954 security.javascriptEnabled option is now available in the YAML configuration file.
Tools

- SERVER-10464\(^{565}\) mongodump can now query oplog.$main and oplog.rs when using --dbpath.
- SERVER-13760\(^{566}\) mongoexport can now handle large timestamps on Windows.

Shell

- SERVER-13865\(^{567}\) Shell now returns correct WriteResult for compatibility-mode upsert with non-OID equality predicate on _id field.
- SERVER-13037\(^{568}\) Fixed typo in error message for “compatibility mode”.

Internal Code

- SERVER-13794\(^{569}\) Fixed: “Unused snapshot history consuming significant heap space”
- SERVER-13446\(^{570}\) Removed Solaris builds dependency on ILLUMOS libc.
- SERVER-14092\(^{571}\) MongoDB upgrade 2.4 to 2.6 check no longer returns an error in internal collections.
- SERVER-14000\(^{572}\) Added new lsb file location for Debian 7.1

Testing

- SERVER-13723\(^{573}\) Stabilized tags.js after a change in its timeout when it was ported to use write commands.
- SERVER-13494\(^{574}\) Fixed: “setup_multiversion_mongodb.py doesn’t download 2.4.10 because of non-numeric version sorting”
- SERVER-13603\(^{575}\) Fixed: “Test suites with options tests fail when run with --nopreallocj”
- SERVER-13948\(^{576}\) Fixed: “awaitReplication() failures related to getting a config version from master causing test failures”
- SERVER-13839\(^{577}\) Fixed sync2.js failure.
- SERVER-13972\(^{578}\) Fixed connections_opened.js failure.
- SERVER-13712\(^{579}\) Reduced peak disk usage of test suites.
- SERVER-14249\(^{580}\) Added tests for querying oplog via mongodump using --dbpath
- SERVER-10462\(^{581}\) Fixed: “Windows file locking related buildbot failures”

\(^{565}\)https://jira.mongodb.org/browse/SERVER-10464
^{566}\)https://jira.mongodb.org/browse/SERVER-13760
^{567}\)https://jira.mongodb.org/browse/SERVER-13865
^{568}\)https://jira.mongodb.org/browse/SERVER-13037
^{569}\)https://jira.mongodb.org/browse/SERVER-13794
^{570}\)https://jira.mongodb.org/browse/SERVER-13446
^{571}\)https://jira.mongodb.org/browse/SERVER-14092
^{572}\)https://jira.mongodb.org/browse/SERVER-14000
^{573}\)https://jira.mongodb.org/browse/SERVER-13723
^{574}\)https://jira.mongodb.org/browse/SERVER-13494
^{575}\)https://jira.mongodb.org/browse/SERVER-13603
^{576}\)https://jira.mongodb.org/browse/SERVER-13948
^{577}\)https://jira.mongodb.org/browse/SERVER-13839
^{578}\)https://jira.mongodb.org/browse/SERVER-13972
^{579}\)https://jira.mongodb.org/browse/SERVER-13712
^{580}\)https://jira.mongodb.org/browse/SERVER-14249
^{581}\)https://jira.mongodb.org/browse/SERVER-10462
2.6.1 – Changes

Stability  SERVER-13739 Repair database failure can delete database files

Build and Packaging

• SERVER-13287 Addition of debug symbols has doubled compile time
• SERVER-13563 Upgrading from 2.4.x to 2.6.0 via yum clobbers configuration file
• SERVER-13691 yum and apt “stable” repositories contain release candidate 2.6.1-rc0 packages
• SERVER-13515 Cannot install MongoDB as a service on Windows

Querying

• SERVER-13066 Negations over multikey fields do not use index
• SERVER-13495 Concurrent GETMORE and KILLCURSORS operations can cause race condition and server crash
• SERVER-13503 The $where operator should not be allowed under $elemMatch
• SERVER-13537 Large skip and and limit values can cause crash in blocking sort stage
• SERVER-13557 Incorrect negation of $elemMatch value in 2.6
• SERVER-13562 Queries that use tailable cursors do not stream results if skip() is applied
• SERVER-13566 Using the OplogReplay flag with extra predicates can yield incorrect results
• SERVER-13611 Missing sort order for compound index leads to unnecessary in-memory sort
• SERVER-13618 Optimization for sorted $in queries not applied to reverse sort order
• SERVER-13661 Increase the maximum allowed depth of query objects
• SERVER-13664 Query with $elemMatch using a compound multikey index can generate incorrect results
• SERVER-13677 Query planner should traverse through $all while handling $elemMatch object predicates
• SERVER-13766 Dropping index or collection while $or query is yielding triggers fatal assertion
Geospatial

- SERVER-13666$\text{near}$ queries with out-of-bounds points in legacy format can lead to crashes
- SERVER-13540$\text{geoNear}$ command no longer returns distance in radians for legacy point
- SERVER-13486$\text{geoNear}$ command can create too large BSON objects for aggregation.

Replication

- SERVER-13500Changing replica set configuration can crash running members
- SERVER-13589Background index builds from a 2.6.0 primary fail to complete on 2.4.x secondaries
- SERVER-13620Replicated data definition commands will fail on secondaries during background index build
- SERVER-13496Creating index with same name but different spec in mixed version replicaset can abort replication

Sharding

- SERVER-12638Initial sharding with hashed shard key can result in duplicate split points
- SERVER-13518The $\text{_id}$ field is no longer automatically generated by $\text{mongos}$ when missing
- SERVER-13777Migrated ranges waiting for deletion do not report cursors still open

Security

- SERVER-9358Log rotation can overwrite previous log files
- SERVER-13644Sensitive credentials in startup options are not redacted and may be exposed
- SERVER-13441Inconsistent error handling in user management shell helpers

Write Operations

- SERVER-13466Error message in collection creation failure contains incorrect namespace
- SERVER-13499Yield policy for batch-inserts should be the same as for batch-updates/deletes
- SERVER-13516Array updates on documents with more than 128 BSON elements may crash $\text{mongod}$

---

600https://jira.mongodb.org/browse/SERVER-13666
601https://jira.mongodb.org/browse/SERVER-13540
602https://jira.mongodb.org/browse/SERVER-13486
603https://jira.mongodb.org/browse/SERVER-13500
604https://jira.mongodb.org/browse/SERVER-13589
605https://jira.mongodb.org/browse/SERVER-13620
606https://jira.mongodb.org/browse/SERVER-13496
607https://jira.mongodb.org/browse/SERVER-12638
608https://jira.mongodb.org/browse/SERVER-13518
609https://jira.mongodb.org/browse/SERVER-13777
610https://jira.mongodb.org/browse/SERVER-9358
611https://jira.mongodb.org/browse/SERVER-13644
612https://jira.mongodb.org/browse/SERVER-13441
613https://jira.mongodb.org/browse/SERVER-13466
614https://jira.mongodb.org/browse/SERVER-13499
615https://jira.mongodb.org/browse/SERVER-13516

---

12.2. Previous Stable Releases
2.6.10 – May 19, 2015

- Improve user cache invalidation enforcement on mongos SERVER-11980
- Provide correct rollbacks for collection creation SERVER-18211
- Allow user inserts into the system.profile collection SERVER-18211
- Fix to query system to ensure non-negation predicates get chosen over negation predicates for multikey index bounds construction SERVER-18364
- All issues closed in 2.6.10

2.6.9 – March 24, 2015

- Resolve connection handling related crash with mongos instances SERVER-17441
- Add server parameter to configure idle cursor timeout SERVER-8188
- Remove duplicated (orphan) documents from aggregation pipelines with _id queries in sharded clusters SERVER-17426
- Fixed crash in geoNear queries with multiple 2dsphere indexes SERVER-14723
- All issues closed in 2.6.9

2.6.8 – February 25, 2015

- Add listCollections command functionality to 2.6 shell and client SERVER-17087
- copydb/clone commands can crash the server if a primary steps down SERVER-16599
- Secondary fasserts trying to replicate an index SERVER-16274
- Query optimizer should always use equality predicate over unique index when possible SERVER-15802
- All issues closed in 2.6.8

2.6.7 – January 13, 2015

- Decreased mongos memory footprint when shards have several tags SERVER-16683
- Removed check for shard version if the primary server is down SERVER-16237
• Fixed: `/etc/init.d/mongod` startup script failure with dirname message SERVER-16081

• Fixed: `mongos` can cause shards to hit the in-memory sort limit by requesting more results than needed SERVER-14306

• All issues closed in 2.6.7

2.6.6 – December 09, 2014

• Fixed: Evaluating candidate query plans with concurrent writes on same collection may crash `mongod` SERVER-15580

• Fixed: 2.6 `mongod` crashes with segfault when added to a 2.8 replica set with 12 or more members SERVER-16107

• Fixed: `$regex`, `$in` and `$sort` with index returns too many results SERVER-15696

• Change: `moveChunk` will fail if there is data on the target shard and a required index does not exist. SERVER-12472

• Primary should abort if encountered problems writing to the oplog SERVER-12058

• All issues closed in 2.6.6

2.6.5 – October 07, 2014

• `$rename` now uses correct dotted source paths SERVER-15029

• Partially written journal last section does not affect recovery SERVER-15111

• Explicitly zero `.ns` files on creation SERVER-15369

• Plan ranker will no longer favor intersection plans if predicate generates empty range index scan SERVER-14961

• Generate Community and Enterprise packages for SUSE 11 SERVER-10642

• All issues closed in 2.6.5

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

---

633 https://jira.mongodb.org/browse/SERVER-16081
634 https://jira.mongodb.org/browse/SERVER-14306
635 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%22%20%22AND%20project%20%3D%20SERVER
636 https://jira.mongodb.org/browse/SERVER-15580
637 https://jira.mongodb.org/browse/SERVER-16107
638 https://jira.mongodb.org/browse/SERVER-15696
639 https://jira.mongodb.org/browse/SERVER-12472
640 https://jira.mongodb.org/browse/SERVER-12058
641 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%22%20%22AND%20project%20%3D%20SERVER
642 https://jira.mongodb.org/browse/SERVER-15029
643 https://jira.mongodb.org/browse/SERVER-15111
644 https://jira.mongodb.org/browse/SERVER-15369
645 https://jira.mongodb.org/browse/SERVER-14961
646 https://jira.mongodb.org/browse/SERVER-10642
647 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%22%20%22AND%20project%20%3D%20SERVER
648 https://jira.mongodb.org/browse/SERVER-14738
• Check the size of the split point before performing a manual split chunk operation SERVER-14431.

• 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.

• Allow read from secondaries when both audit and authorization are enabled in a sharded cluster SERVER-14170.

• All issues closed in 2.6.4.

2.6.3 – June 19, 2014

• Equality queries on _id with projection may return no results on sharded collections SERVER-14302.

• Equality queries on _id with projection on _id may return orphan documents on sharded collections SERVER-14304.

• All issues closed in 2.6.3.

2.6.2 – June 16, 2014

• Query plans with differing performance can tie during plan ranking SERVER-13675.

• mongod may terminate if x.509 authentication certificate is invalid SERVER-13753.

• Temporary map/reduce collections are incorrectly replicated to secondaries SERVER-13981.

• mongos incorrectly targets multiple shards for nested field shard key predicates SERVER-14138.

• rs.stepDown() during mapReduce causes fassert when writing to op log SERVER-14186.

• All issues closed in 2.6.2.

2.6.1 – May 5, 2014

• Fix to install MongoDB service on Windows with the --install option SERVER-13515.

• Allow direct upgrade from 2.4.x to 2.6.0 via yum SERVER-13563.

• Fix issues with background index builds on secondaries: SERVER-13589 and SERVER-13620.

• Redact credential information passed as startup options SERVER-13644.
• 2.6.1 Changelog (page 846).

• All issues closed in 2.6.1.

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.

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 500) 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.

667https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.6.1%22%20AND%20project%20%3D%20SERVER
– `$min` and `$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 861)

**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 857)

**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 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 324) that provides the ability to create custom *User-Defined Roles* (page 325) 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 361) as well as for *internal authentication* (page 363) of sharded and/or replica set cluster members. x.509 authentication is only available for deployments using SSL.

• Enhanced SSL Support:
  – *Rolling upgrades of clusters* (page 350) to use SSL.
  – *MongoDB Tools* (page 350) support connections to `mongod` and `mongos` instances using SSL connections.
  – *Prompt for passphrase* (page 345) by `mongod` or `mongos` at startup.
Require the use of strong 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 327). To enable the interface, see enabled.

See also:
New Authorization Model (page 859), SSL Certificate Hostname Validation (page 860), and Security Checklist (page 431).

Query Engine Improvements

- MongoDB can now use index intersection (page 510) to fulfill queries supported by more than one index.
- Index Filters (page 69) 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 68) cached by the query optimizer.
- MongoDB can now use count() with hint(). See count() for details.

Improvements

Geospatial Enhancements

- 2dsphere indexes version 2 (page 496).
- Support for MultiPoint (page 554), MultiLineString (page 555), MultiPolygon (page 555), and GeometryCollection (page 556).
- Support for geospatial query clauses in $or expressions.

See also:
2dsphere Index Version 2 (page 860), $maxDistance Changes (page 863), Deprecated SuniqueDocs (page 863), Stronger Validation of Geospatial Queries (page 863)

Index Build Enhancements

- Background index build (page 508) 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.

12.2. Previous Stable Releases
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 725).
- 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 594), 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 89) 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 49) 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 330) capability for `mongod` and `mongos` instances. See Auditing (page 330) 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 369) and Authenticate Using SASL and LDAP with ActiveDirectory (page 366) 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 866) 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 861)

Additional Information

Changes Affecting Compatibility

Compatibility Changes in MongoDB 2.6  The following 2.6 changes can affect the compatibility with older versions of MongoDB. See *Release Notes for MongoDB 2.6* (page 827) 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 78) directly into the method rather than with a separate `getLastError` command to provide `safe writes` (page 78)
whether run interactively in the mongo shell or non-interactively in a script. In previous versions, these methods exhibited a “fire-and-forget” behavior.

- 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.

**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:

```javascript
for (var i = 1; i <= 1000000; i++) {
  db.test.insert({ x: i });
}
```

In MongoDB 2.6, replace with Bulk() operation:

```javascript
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 852), Bulk(), Bulk.execute(), db.collection.initializeUnorderedBulkOp(), db.collection.initializeOrderedBulkOp()
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:

```javascript
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 851), `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 324) 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 866).

**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 **before** upgrading the MongoDB binaries.

**See also:**

*Security Improvements* (page 852)
SSL Certificate Hostname Validation

Description The 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 SSL and neither the CN nor any of the SAN entries of your current SSL certificates match the hostnames, upgrading to version 2.6 will cause the 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 SSL certificates on other servers in the cluster.
- `mongo` shell, *MongoDB tools that support SSL* (page 350), 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 496). 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 496)

Log Messages

Timestamp Format Change

Description Each message now starts with the timestamp format given in *Time Format Changes* (page 866). 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:
  ```javascript
  { $set: [ ] }
  ```
- Update operators (e.g. `$set`) cannot repeat in the update statement. For example, the following expression is invalid:
  ```javascript
  { $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 852) for the changes to the write operation protocol, and `Insert and Update Improvements` (page 851) 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 505) 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 506) 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 68) 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

$\text{maxDistance} \text{ Changes}

Description

- For $\text{near}$ queries on GeoJSON data, if the queries specify a $\text{maxDistance}$, $\text{maxDistance}$ must be inside of the $\text{near}$ document.

  In previous version, $\text{maxDistance}$ could be either inside or outside the $\text{near}$ document.

- $\text{maxDistance}$ must be a positive value.

Solution

- Update any existing $\text{near}$ queries on GeoJSON data that currently have the $\text{maxDistance}$ outside the $\text{near}$ document

- Update any existing queries where $\text{maxDistance}$ is a negative value.

Deprecated $\text{uniqueDocs}$

Description

MongoDB 2.6 deprecates $\text{uniqueDocs}$, 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

$\text{not}$ Query Behavior Changes

Description

- Queries with $\text{not}$ 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:

```json
{ _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:

```javascript
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 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:

```json
{ "_id": 1, "a" : [ [ 1, 2, 5 ] ] }
```

In 2.6, the following $elemMatch query does *not* match the document:

```javascript
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 866) 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 866).

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` 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` 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. Other Resources

- All backwards incompatible changes (JIRA).
- Release Notes for MongoDB 2.6 (page 827).
- Upgrade MongoDB to 2.6 (page 866) for the upgrade process.

Some changes in 2.6 can affect compatibility 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 856) for a detailed list of compatibility changes.

See also:
- All Backwards incompatible changes (JIRA).

Upgrade Process

Upgrade MongoDB to 2.6 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 866), the procedure for upgrading sharded clusters (page 868), and the considerations for reverting to 2.4 after running 2.6 (page 871).

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 892) for the procedure to upgrade from 2.2 to 2.4.

If you use MongoDB Cloud Manager 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 856) for an explanation of these changes. Resolve all incompatibilities in your deployment before continuing.
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.

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 870).

Before beginning the upgrade process for a deployment that uses authentication and authorization:

- Ensure that at least one user exists in the admin database.
- 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.

See Upgrade User Authorization Data to 2.6 Format (page 870) 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:

```bash
sudo apt-get update
sudo apt-get install mongodb-org
```

For Red Hat Enterprise, CentOS, Fedora, or Amazon Linux:

```bash
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 16), RHEL (page 7), Debian (page 19), or other Linux Systems (page 22) 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 892) 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. 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.

673 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 892) 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 580). 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 736) for a complete list of sharding commands. Note, however, that not all commands on the Sharding Reference (page 736) 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 684) in the sharded cluster, as described in Disable the Balancer (page 717).
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 869) 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 718).

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 225) 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

Upgrade User Authorization Data to 2.6 Format  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 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.

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.

674https://www.mongodb.com/products/consulting?jmp=docs#major_version_upgrade
For a sharded cluster, do not run the upgrade process directly against the config servers (page 670). 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 412).

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 412).

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 291) 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 Before you attempt any downgrade, familiarize yourself with the content of this document, particularly the Downgrade Recommendations and Checklist (page 871) and the procedure for downgrading sharded clusters (page 876).

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 874) before downgrading MongoDB 2.6 to 2.4.
- Remove or downgrade version 2 2dsphere indexes (page 875) before downgrading MongoDB 2.6 to 2.4.
- Downgrade 2.6 User Authorization Model (page 872). 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 876).
- To downgrade replica sets, see Downgrade a 2.6 Replica Set (page 875).
- To downgrade a standalone MongoDB instance, see Downgrade 2.6 Standalone mongod Instance (page 875).

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 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:

```json
{ 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 412).

Step 2: Create a role with required privileges.

Using the db.createRole method, create a role (page 325) with the required privileges.

```javascript
use admin
db.createRole(
    { role: "downgradeAuthRole", privileges: [ 
        { resource: { db: "admin", collection: "system.new_users" }, actions: [ "find", "insert", "update" ] }
    ] })
```
Step 3: Create a user with the new role. Create a user and assign the user the `downgradeRole`.

```javascript
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.

```javascript
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 872) for details.

Step 2: Create backup of 2.6 `admin.system.users` collection. Copy all documents in the `admin.system.users` (page 291) collection to the `admin.system.new_users` collection:

```javascript
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.

```javascript
db.getSiblingDB("admin").system.version.update({
    _id: "authSchema",
    $set: { currentVersion: 2 } })
```

12.2. Previous Stable Releases 873
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 421) action, which is granted by `userAdminAnyDatabase` (page 412) and `hostManager` (page 410) 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
    }
}```
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.

```json
{
  "v" : 2,
  "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. 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 225) 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 580) 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()
- 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 736) for a complete list of sharding commands. Note, however, that not all commands on the Sharding Reference (page 736) 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 684) in the sharded cluster, as described in Disable the Balancer (page 717).
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 718).
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 225) 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 866) for full upgrade instructions.

Download

To download MongoDB 2.6, go to the downloads page.

Other Resources

- All JIRA issues resolved in 2.6.
- All Third Party License Notices.

12.2.2 Release Notes for MongoDB 2.4

March 19, 2013

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

2.4.14

- Packaging: Init script sets process ulimit to different value compared to documentation (SERVER-17780)
- Security: Compute BinData length in v8 (SERVER-17647)

---

676 http://www.mongodb.org/downloads
677 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%2C+rc1%22%2C%222.6.0-rc2%22%2C%222.6.0-rc3%22+%2C%222.6.0-rc4%22+%222.6.0-rc5%22+%222.6.0-rc6%22%2C+rc8%22%2C%222.6.0-rc9%22%2C+rc10%22%2C%222.6.0+rc11%22%2C+rc12%22%2C%222.6.0-rc13%22%2C+rc14%22%2C%222.6.0-rc15%22%2C+rc16%22%2C%222.6.0-rc17%22%2C+rc18%22%2C+rc19%22%2C+rc20%22%2C+rc21%22%2C%222.6.0-rc22%22%2C%222.6.0-rc23%22%2C+rc24%22%2C%222.6.0-rc25%22%2C%222.6.0-rc26%22%2C+rc27%22%2C%222.6.0-rc28%22%2C%222.6.0-rc29%22%2C%222.6.0-rc30%22%2C%222.6.0-rc31%22%2C+rc32%22%2C%222.6.0-rc33%22%2C+rc34%22%2C%222.6.0-rc35%22%2C%222.6.0-rc36%22%2C%222.6.0-rc37%22%2C+rc38%22%2C%222.6.0-rc39%22%2C+rc40%22%2C%222.6.0-rc41%22%2C+rc42%22%2C%222.6.0-rc43%22%2C%222.6.0-rc44%22+);
678 https://github.com/mongodb/mongo/blob/v2.6/dist/src/THIRD-PARTY-NOTICES
679 https://jira.mongodb.org/browse/SERVER-17780
680 https://jira.mongodb.org/browse/SERVER-17647
• Build: Upgrade PCRE Version from 8.30 to Latest (SERVER-17252681)

2.4.13 - Changes

• Security: Enforce BSON BinData length validation (SERVER-17278682)
• Security: Disable SSLv3 ciphers (SERVER-15673683)
• Networking: Improve BSON validation (SERVER-17264684)

2.4.12 - Changes

• Sharding: Sharded connection cleanup on setup error can crash mongos (SERVER-15056685)
• Sharding: “type 7” (OID) error when acquiring distributed lock for first time (SERVER-13616686)
• Storage: explicitly zero .ns files on creation (SERVER-15369687)
• Storage: partially written journal last section causes recovery to fail (SERVER-15111688)

2.4.11 - Changes

• Security: Potential information leak (SERVER-14268689)
• Replication: _id with $prefix field causes replication failure due to unvalidated insert (SERVER-12209690)
• Sharding: Invalid access: seg fault in SplitChunkCommand::run (SERVER-14342691)
• Indexing: Creating descending index on _id can corrupt namespace (SERVER-14832692)
• Text Search: Updates to documents with text-indexed fields may lead to incorrect entries (SERVER-14738693)
• Build: Add SCons flag to override treating all warnings as errors (SERVER-13724694)
• Packaging: Fix mongodb enterprise 2.4 init script to allow multiple processes per host (SERVER-14336695)
• JavaScript: Do not store native function pointer as a property in function prototype (SERVER-14254696)

2.4.10 - Changes

• Indexes: Fixed issue that can cause index corruption when building indexes concurrently (SERVER-12990697)
• Indexes: Fixed issue that can cause index corruption when shutting down secondary node during index build (SERVER-12956698)

https://jira.mongodb.org/browse/SERVER-17252
https://jira.mongodb.org/browse/SERVER-17278
https://jira.mongodb.org/browse/SERVER-15673
https://jira.mongodb.org/browse/SERVER-17264
https://jira.mongodb.org/browse/SERVER-15056
https://jira.mongodb.org/browse/SERVER-13616
https://jira.mongodb.org/browse/SERVER-15369
https://jira.mongodb.org/browse/SERVER-15111
https://jira.mongodb.org/browse/SERVER-14209
https://jira.mongodb.org/browse/SERVER-14342
https://jira.mongodb.org/browse/SERVER-14833
https://jira.mongodb.org/browse/SERVER-14738
https://jira.mongodb.org/browse/SERVER-13724
https://jira.mongodb.org/browse/SERVER-14336
https://jira.mongodb.org/browse/SERVER-14254
https://jira.mongodb.org/browse/SERVER-12990
https://jira.mongodb.org/browse/SERVER-12956
• Indexes: Mongod now recognizes incompatible “future” text and geo index versions and exits gracefully (SERVER-12914699)

• Indexes: Fixed issue that can cause secondaries to fail replication when building the same index multiple times concurrently (SERVER-12662700)

• Indexes: Fixed issue that can cause index corruption on the tenth index in a collection if the index build fails (SERVER-12481701)

• Indexes: Introduced versioning for text and geo indexes to ensure backwards compatibility (SERVER-12175702)

• Indexes: Disallowed building indexes on the system.indexes collection, which can lead to initial sync failure on secondaries (SERVER-10231703)

• Sharding: Avoid frequent immediate balancer retries when config servers are out of sync (SERVER-12908704)

• Sharding: Add indexes to locks collection on config servers to avoid long queries in case of large numbers of collections (SERVER-12548705)

• Sharding: Fixed issue that can corrupt the config metadata cache when sharding collections concurrently (SERVER-12515706)

• Sharding: Don’t move chunks created on collections with a hashed shard key if the collection already contains data (SERVER-9259707)

• Replication: Fixed issue where node appears to be down in a replica set during a compact operation (SERVER-12264708)

• Replication: Fixed issue that could cause delays in elections when a node is not vetoing an election (SERVER-12170709)

• Replication: Step down all primaries if multiple primaries are detected in replica set to ensure correct election result (SERVER-10793710)

• Replication: Upon clock skew detection, secondaries will switch to sync directly from the primary to avoid sync cycles (SERVER-8375711)

• Runtime: The SIGXCPU signal is now caught and mongod writes a log message and exits gracefully (SERVER-12034712)

• Runtime: Fixed issue where mongod fails to start on Linux when /sys/dev/block directory is not readable (SERVER-9248713)

• Windows: No longer zero-fill newly allocated files on systems other than Windows 7 or Windows Server 2008 R2 (SERVER-8480714)

699 https://jira.mongodb.org/browse/SERVER-12914
700 https://jira.mongodb.org/browse/SERVER-12662
701 https://jira.mongodb.org/browse/SERVER-12481
702 https://jira.mongodb.org/browse/SERVER-12175
703 https://jira.mongodb.org/browse/SERVER-10231
704 https://jira.mongodb.org/browse/SERVER-12908
705 https://jira.mongodb.org/browse/SERVER-12548
706 https://jira.mongodb.org/browse/SERVER-12515
707 https://jira.mongodb.org/browse/SERVER-9259
708 https://jira.mongodb.org/browse/SERVER-12264
709 https://jira.mongodb.org/browse/SERVER-12170
710 https://jira.mongodb.org/browse/SERVER-10793
711 https://jira.mongodb.org/browse/SERVER-8375
712 https://jira.mongodb.org/browse/SERVER-12034
713 https://jira.mongodb.org/browse/SERVER-9248
714 https://jira.mongodb.org/browse/SERVER-8480
• GridFS: Chunk size is decreased to 255 KB (from 256 KB) to avoid overhead with usePowerOf2Sizes option (SERVER-13331715)
• SNMP: Fixed MIB file validation under smilint (SERVER-12487716)
• Shell: Fixed issue in V8 memory allocation that could cause long-running shell commands to crash (SERVER-11871717)
• Shell: Fixed memory leak in the md5sumFile shell utility method (SERVER-11560718)

Previous Releases
• All 2.4.9 improvements719.
• All 2.4.8 improvements720.
• All 2.4.7 improvements721.
• All 2.4.6 improvements722.
• All 2.4.5 improvements723.
• All 2.4.4 improvements724.
• All 2.4.3 improvements725.
• All 2.4.2 improvements726.
• All 2.4.1 improvements727.

2.4.14 – April 28, 2015
• Init script sets process ulimit to different value compared to documentation SERVER-17780728
• Compute BinData length in v8 SERVER-17647729
• Upgrade PCRE Version from 8.30 to Latest SERVER-17252730
• 2.4.14 Changelog (page 877).
• All 2.4.14 improvements731.

715https://jira.mongodb.org/browse/SERVER-13331
716https://jira.mongodb.org/browse/SERVER-12487
717https://jira.mongodb.org/browse/SERVER-11871
718https://jira.mongodb.org/browse/SERVER-11560
719https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
720https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
721https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
722https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.6%22%20AND%20project%20%3D%20SERVER
723https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
724https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
725https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.3%22%20AND%20project%20%3D%20SERVER
726https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.2%22%20AND%20project%20%3D%20SERVER
727https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.1%22%20AND%20project%20%3D%20SERVER
728https://jira.mongodb.org/browse/SERVER-17780
729https://jira.mongodb.org/browse/SERVER-17647
730https://jira.mongodb.org/browse/SERVER-17252
731https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.14%22%20AND%20project%20%3D%20SERVER
2.4.13 – February 25, 2015

- Enforce BSON BinData length validation SERVER-17278
- Disable SSLv3 ciphers SERVER-15673
- Improve BSON validation SERVER-17264
- 2.4.13 Changelog (page 878).
- All 2.4.13 improvements.

2.4.12 – October 16, 2014

- Partially written journal last section causes recovery to fail SERVER-15111.
- Explicitly zero .ns files on creation SERVER-15369.
- 2.4.12 Changelog (page 878).
- All 2.4.12 improvements.

2.4.11 – August 18, 2014

- Fixed potential information leak: SERVER-14268.
- Resolved issue were an _id with a $prefix field caused replication failure due to unvalidated insert SERVER-12209.
- Addressed issue where updates to documents with text-indexed fields could lead to incorrect entries SERVER-14738.
- Resolved issue where creating descending index on _id could corrupt namespace SERVER-14833.
- 2.4.11 Changelog (page 878).
- All 2.4.11 improvements.

2.4.10 – April 4, 2014

- Performs fast file allocation on Windows when available SERVER-8480.
- Start elections if more than one primary is detected SERVER-10793.

---

732 https://jira.mongodb.org/browse/SERVER-17278
733 https://jira.mongodb.org/browse/SERVER-15673
734 https://jira.mongodb.org/browse/SERVER-17264
735 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.13%22%20AND%20project%20%3D%20SERVER
736 https://jira.mongodb.org/browse/SERVER-15111
737 https://jira.mongodb.org/browse/SERVER-15369
738 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.12%22%20AND%20project%20%3D%20SERVER
739 https://jira.mongodb.org/browse/SERVER-14268
740 https://jira.mongodb.org/browse/SERVER-12209
741 https://jira.mongodb.org/browse/SERVER-14738
742 https://jira.mongodb.org/browse/SERVER-14833
743 https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.11%22%20AND%20project%20%3D%20SERVER
744 https://jira.mongodb.org/browse/SERVER-8480
745 https://jira.mongodb.org/browse/SERVER-10793
• Changes to allow safe downgrading from v2.6 to v2.4 SERVER-12914\textsuperscript{746}, SERVER-12175\textsuperscript{747}.
• Fixes for edge cases in index creation SERVER-12481\textsuperscript{748}, SERVER-12956\textsuperscript{749}.
• 2.4.10 Changelog (page 878).
• All 2.4.10 improvements\textsuperscript{750}.

2.4.9 – January 10, 2014

• Fix for instances where \texttt{mongos} incorrectly reports a successful write SERVER-12146\textsuperscript{751}.
• Make non-primary read preferences consistent with \texttt{slaveOK} versioning logic SERVER-11971\textsuperscript{752}.
• Allow new sharded cluster connections to read from secondaries when primary is down SERVER-7246\textsuperscript{753}.
• All 2.4.9 improvements\textsuperscript{754}.

2.4.8 – November 1, 2013

• Increase future compatibility for 2.6 authorization features SERVER-11478\textsuperscript{755}.
• Fix \texttt{dbhash} cache issue for config servers SERVER-11421\textsuperscript{756}.
• All 2.4.8 improvements\textsuperscript{757}.

2.4.7 – October 21, 2013

• Fixed over-aggressive caching of V8 Isolates SERVER-10596\textsuperscript{758}.
• Removed extraneous initial count during mapReduce SERVER-9907\textsuperscript{759}.
• Cache results of dbhash command SERVER-11021\textsuperscript{760}.
• Fixed memory leak in aggregation SERVER-10554\textsuperscript{761}.
• All 2.4.7 improvements\textsuperscript{762}.

\textsuperscript{746}https://jira.mongodb.org/browse/SERVER-12914
\textsuperscript{747}https://jira.mongodb.org/browse/SERVER-12175
\textsuperscript{748}https://jira.mongodb.org/browse/SERVER-12481
\textsuperscript{749}https://jira.mongodb.org/browse/SERVER-12956
\textsuperscript{750}https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.10%22%20AND%20project%20%3D%20SERVER
\textsuperscript{751}https://jira.mongodb.org/browse/SERVER-12146
\textsuperscript{752}https://jira.mongodb.org/browse/SERVER-11971
\textsuperscript{753}https://jira.mongodb.org/browse/SERVER-7246
\textsuperscript{754}https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.9%22%20AND%20project%20%3D%20SERVER
\textsuperscript{755}https://jira.mongodb.org/browse/SERVER-11478
\textsuperscript{756}https://jira.mongodb.org/browse/SERVER-11421
\textsuperscript{757}https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.8%22%20AND%20project%20%3D%20SERVER
\textsuperscript{758}https://jira.mongodb.org/browse/SERVER-10596
\textsuperscript{759}https://jira.mongodb.org/browse/SERVER-9907
\textsuperscript{760}https://jira.mongodb.org/browse/SERVER-11021
\textsuperscript{761}https://jira.mongodb.org/browse/SERVER-10554
\textsuperscript{762}https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.7%22%20AND%20project%20%3D%20SERVER
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-10478763.
- Fix for C++ client shutdown issues SERVER-8891764.
- Improved replication robustness in presence of high network latency SERVER-10085765.
- Improved Solaris support SERVER-9832766, SERVER-9786767, and SERVER-7080768.
- All 2.4.6 improvements769.

2.4.5 – July 3, 2013

- Fix for CVE-2013-4650 Improperly grant user system privileges on databases other than local SERVER-9983770.
- Fix to prevent identical background indexes from being built SERVER-9856772.
- Config server performance improvements SERVER-9864773 and SERVER-5442774.
- Improved initial sync resilience to network failure SERVER-9853775.
- All 2.4.5 improvements776.

2.4.4 – June 4, 2013

- Performance fix for Windows version SERVER-9721777
- Fix for config upgrade failure SERVER-9661778.
- Migration to Cyrus SASL library for MongoDB Enterprise SERVER-8813779.
- All 2.4.4 improvements780.

2.4.3 – April 23, 2013

- Fix for mongo shell ignoring modified object’s _id field SERVER-9385781.

763https://jira.mongodb.org/browse/SERVER-10478
764https://jira.mongodb.org/browse/SERVER-8891
765https://jira.mongodb.org/browse/SERVER-10085
766https://jira.mongodb.org/browse/SERVER-9832
767https://jira.mongodb.org/browse/SERVER-9786
768https://jira.mongodb.org/browse/SERVER-7080
769https://jira.mongodb.org/browse/SERVER-9832
770https://jira.mongodb.org/browse/SERVER-9983
771https://jira.mongodb.org/browse/SERVER-9878
772https://jira.mongodb.org/browse/SERVER-9856
773https://jira.mongodb.org/browse/SERVER-9864
774https://jira.mongodb.org/browse/SERVER-5442
775https://jira.mongodb.org/browse/SERVER-9853
776https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.5%22%20AND%20project%20%3D%20SERVER
777https://jira.mongodb.org/browse/SERVER-9721
778https://jira.mongodb.org/browse/SERVER-9661
779https://jira.mongodb.org/browse/SERVER-8813
780https://jira.mongodb.org/issues/?jql=fixVersion%20%3D%20%222.4.4%22%20AND%20project%20%3D%20SERVER
781https://jira.mongodb.org/browse/SERVER-9385

12.2. Previous Stable Releases
• Fix for race condition in log rotation SERVER-4739782.
• Fix for copydb command with authorization in a sharded cluster SERVER-9093783.
• All 2.4.3 improvements784.

2.4.2 – April 17, 2013

• Several V8 memory leak and performance fixes SERVER-9267785 and SERVER-9230786.
• Fix for upgrading sharded clusters SERVER-9125787.
• Fix for high volume connection crash SERVER-9014788.
• All 2.4.2 improvements789

2.4.1 – April 17, 2013

• Fix for losing index changes during initial sync SERVER-9087790.
• All 2.4.1 improvements791.

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 500) for more information.

Geospatial Support Enhancements

• Add new 2dsphere index (page 496). The new index supports GeoJSON objects Point, LineString, and Polygon. See 2dsphere Indexes (page 496) 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 502) 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 674).

http://geojson.org/geojson-spec.html
Improvements to the Aggregation Framework

- Improve support for geospatial queries. See the `$geoWithin` operator and the `$geoNear` pipeline stage.
- Improve sort efficiency when the `$sort` stage immediately precedes a `$limit` in the pipeline.
- Add new operators `$millisecond` and `$concat` and modify how `$min` operator processes null values.

Changes to Update Operators

- Add new `$setOnInsert` operator for use with `upsert`.
- Enhance functionality of the `$push` operator, supporting its use with the `$each`, the `$sort`, and the `$slice` modifiers.

Additional Limitations for Map-Reduce and `$where` Operations

The `mapReduce` command, `group` command, and the `$where` operator expressions cannot access certain global functions or properties, such as `db`, that are available in the `mongo` shell. See the individual command or operator for details.

Improvements to `serverStatus` Command

Provide additional metrics and customization for the `serverStatus` command. See `db.serverStatus()` and `serverStatus` for more information.

Security Enhancements

- Introduce a role-based access control system User Privileges 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 342).

For more information on security and risk management strategies, see `MongoDB Security Practices and Procedures` (page 317).

Performance Improvements

V8 JavaScript Engine

JavaScript Changes in MongoDB 2.4 Consider the following impacts of V8 JavaScript Engine (page 885) 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.

793 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\(^794\), abbreviated as ES5, adds many new language features, including:

- standardized JSON\(^795\),
- strict mode\(^796\),
- function.bind()\(^797\),
- array extensions\(^798\), 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](https://jira.mongodb.org/browse/SERVER-8216), [SERVER-8223](https://jira.mongodb.org/browse/SERVER-8223), [SERVER-8215](https://jira.mongodb.org/browse/SERVER-8215), and [SERVER-8214](https://jira.mongodb.org/browse/SERVER-8214) for more information.

For objects that have not been returned from MongoDB queries, the features work as expected.

**Removed Non-Standard SpiderMonkey Features**  V8 does **not** support the following non-standard SpiderMonkey\(^803\) JavaScript extensions, previously supported by MongoDB’s use of SpiderMonkey as its JavaScript engine.

**E4X Extensions**  V8 does not support the non-standard E4X\(^804\) extensions. E4X provides a native XML\(^805\) 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()
Destructuring Assignment  V8 does not support the non-standard destructuring assignments. Destructuring assignment “extract[s] data from arrays or objects using a syntax that mirrors the construction of array and object literals.” - Mozilla docs

Example
The following destructuring assignment is invalid with V8 and throws a SyntaxError:

```javascript
original = [4, 8, 15];
var [b, ,c] = a;  // <= destructuring assignment
print(b)  // 4
print(c)  // 15
```

Iterator(), StopIteration(), and Generators  V8 does not support Iterator(), StopIteration(), and generators.

InternalError()  V8 does not support InternalError(). Use Error() instead.

for each...in Construct  V8 does not support the use of for each...in construct. Use for (var x in y) construct instead.

Example
The following for each (var x in y) construct is invalid with V8:

```javascript
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:

```javascript
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():

```javascript
Object.keys(o).forEach(function (key) {  
  var value = o[key];  
  print(value);  
});
```

---

806 https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7#Destructuring_assignment_(Merge_into_own_page.2Fsection)
808 https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/for_each...in
Array Comprehension  V8 does not support *Array comprehensions*. Use other methods such as the *Array instance methods* `map()`, `filter()`, or `forEach()`.

**Example**

With V8, the following array comprehension is invalid:

```javascript
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()`:

```javascript
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 890) 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":

```javascript
try {
    something()
} catch (err if err instanceof SomeError) {
    print('some error')
} catch (err) {
    print('standard error')
}
```

---

Conditional Function Definition  V8 will produce different outcomes than SpiderMonkey with *conditional function definitions*.

**Example**

The following conditional function definition produces different outcomes in SpiderMonkey versus V8:

```javascript
function test () {
    if (false) {
```

---


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.

```javascript
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.

```javascript
function test () {
    'use strict';
    if (false) {
        function go () {};
    }
}
```

The JavaScript code throws the following syntax error:

```javascript
SyntaxError: In strict mode code, functions can only be declared at top level or immediately within a function.
```

### String Generic Methods

V8 does not support String generics. String generics are a set of methods on the String class that mirror instance methods.

**Example**

The following use of the generic method `String.toLowerCase()` is invalid with V8:

```javascript
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:

```javascript
var name = 'MongoDB';
var lower = name.toLowerCase();
print(name + ' becomes ' + lower);
```

---

811 http://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/
With V8, use the `String` *instance* methods instead of following *generic* methods:

<table>
<thead>
<tr>
<th>String.charAt()</th>
<th>String.quote()</th>
<th>String.toLocaleLowerCase()</th>
</tr>
</thead>
<tbody>
<tr>
<td>String.charCodeAt()</td>
<td>String.replace()</td>
<td>String.toLocaleUpperCase()</td>
</tr>
<tr>
<td>String.concat()</td>
<td>String.search()</td>
<td>String.toUpperCase()</td>
</tr>
<tr>
<td>String.endsWith()</td>
<td>String.slice()</td>
<td>String.trim()</td>
</tr>
<tr>
<td>String.valueOf()</td>
<td>String.split()</td>
<td>String.trimLeft()</td>
</tr>
<tr>
<td>String.lastIndexOf()</td>
<td>String.startsWith()</td>
<td>String.trimRight()</td>
</tr>
<tr>
<td>String.localeCompare()</td>
<td>String.substr()</td>
<td>String.trim()</td>
</tr>
<tr>
<td>String.match()</td>
<td>String.substring()</td>
<td>String()</td>
</tr>
</tbody>
</table>

**Array Generic Methods**  V8 does not support *Array generic methods*[^813]. *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:

```javascript
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:

```javascript
var allEven = arr.every(isEven);
print(allEven);
```

With V8, use the *Array* *instance* methods instead of the following *generic* methods:

<table>
<thead>
<tr>
<th>Array.concat()</th>
<th>Array.lastIndexOf()</th>
<th>Array.slice()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array.every()</td>
<td>Array.map()</td>
<td>Array.some()</td>
</tr>
<tr>
<td>Array.filter()</td>
<td>Array.pop()</td>
<td>Array.sort()</td>
</tr>
<tr>
<td>Array.forEach()</td>
<td>Array.push()</td>
<td>Array.splice()</td>
</tr>
<tr>
<td>Array.indexOf()</td>
<td>Array.reverse()</td>
<td>Array.unshift()</td>
</tr>
<tr>
<td>Array.join()</td>
<td>Array.shift()</td>
<td>Array.toString()</td>
</tr>
</tbody>
</table>

**Array Instance Method `toSource()`**  V8 does not support the *Array* instance method `toSource()`[^814]. Use the *Array* instance method `toString()` instead.

**uneval()**  V8 does not support the non-standard method `uneval()`. Use the standardized `JSON.stringify()`[^815] method instead.

Change default JavaScript engine from SpiderMonkey to V8. The change provides improved concurrency for JavaScript operations, modernized JavaScript implementation, and the removal of non-standard SpiderMonkey features, and affects all JavaScript behavior including the commands `mapReduce`, `group`, and `eval` and the query operator `$where`.

See *JavaScript Changes in MongoDB 2.4* (page 885) 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 508) 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 899) 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 685) in a *sharded cluster* propagates to at least one secondary before the balancer proceeds with its next operation. See *Chunk Migration and Replication* (page 687).

**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 686) 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 33) 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 372) for more information. For drivers that provide support for Kerberos authentication to MongoDB, refer to Driver Support (page 333).

For more information on security and risk management strategies, see MongoDB Security Practices and Procedures (page 317).

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 for more information on platform support.

Upgrade Process

Upgrade MongoDB to 2.4  In the general case, the upgrade from MongoDB 2.2 to 2.4 is a binary-compatible “drop-in” upgrade: shut down the mongod instances and replace them with mongod instances running 2.4. However, before you attempt any upgrade please familiarize yourself with the content of this document, particularly the procedure for upgrading sharded clusters (page 893) and the considerations for reverting to 2.2 after running 2.4 (page 897).

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 893).

• 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 897).

• 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 885) for more information.

Upgrade Standalone mongod Instance to MongoDB 2.4

1. Download binaries of the latest release in the 2.4 series from the MongoDB Download Page. See Install MongoDB (page 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:

816[http://www.mongodb.org/downloads/]
817[http://www.mongodb.org/downloads]
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 \texttt{rs.status()} in the mongo shell.

2. Use the mongo shell method \texttt{rs.stepDown()} to step down the primary to allow the normal failover (page 580) procedure. \texttt{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 \texttt{rs.status()}, shut down the previous primary and replace mongod binary with the 2.4 binary and start the new process.

\textbf{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 pre-defined maintenance window.

---

**Upgrade a Sharded Cluster from MongoDB 2.2 to MongoDB 2.4**

\textbf{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.

---

\textbf{Overview} Upgrading a \textit{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 893) 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 896) for more information.

---

\textbf{Cluster Meta-data Upgrade}

\textbf{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 \textit{config database} (page 737) is at least 4 to 5 times the amount of space currently used by the \textit{config database} (page 737) data files.

  Additionally, ensure that all indexes in the \textit{config database} (page 737) are \{\texttt{v:1}\} indexes. If a critical index is a \{\texttt{v:0}\} index, chunk splits can fail due to known issues with the \{\texttt{v:0}\} format. \{\texttt{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 736) for a complete list of sharding commands. Note, however, that not all commands on the Sharding Reference (page 736) 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 739) and config.collections (page 740) collections. As always, if storage requirements increase, the mongod might need to pre-allocate additional data files. See How can I get information on the storage use of a database? (page 777) for more information.

Meta-data Upgrade Procedure  Changes to the meta-data format for sharded clusters, stored in the config database (page 737), 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 893) 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 737) is at least 4 to 5 times the amount of space currently used by the config database (page 737) data files.

   Additionally, ensure that all indexes in the config database (page 737) 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.getIndexNames().

   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 684) in the sharded cluster, as described in Disable the Balancer (page 717).

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 670) 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 896) 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 : 0 } })
```

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 896) for more information.

7. Re-enable the balancer (page 717). 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 670) 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 894).

To resync the config servers:

1. Turn off the balancer (page 684) 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 893)), 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 893).

Upgrade Sharded Cluster Components  After you have successfully completed the meta-data upgrade process described in Meta-data Upgrade Procedure (page 894), 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 717).

**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:

```javascript
db.records.dropIndex( { loc: "2dsphere" } )
```

```javascript
db.records.dropIndex( "records_text" )
```

```javascript
db.records.ensureIndex( { loc: "2dsphere" } )
```

```javascript
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 893), 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 893), 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 upgrading, exiting
```

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 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

Compatibility and Index Type Changes in MongoDB 2.4  In 2.4 MongoDB includes two new features related to indexes that users upgrading to version 2.4 must consider, particularly with regard to possible downgrade paths. For more information on downgrades, see Downgrade MongoDB from 2.4 to Previous Versions (page 897).

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 897) 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.

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.

Example

If you attempt to add an invalid index in MongoDB 2.4, as in the following:

db.coll.ensureIndex( { field: "1" } )

MongoDB will return the following error document:

```json
{
    "err": "Unknown index plugin '1' in index { field: "1\" }"
    "code": 16734,
    "n": <number>,
    "connectionId": <number>,
    "ok": 1
}
```

See Upgrade MongoDB to 2.4 (page 892) for full upgrade instructions.

Other Resources

- MongoDB Downloads

818 The data file version (i.e. pdfile version) is independent and unrelated to the release version of MongoDB.
819 https://www.mongodb.com/products/consulting#jmp=docs#major_version_upgrade
820 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.
821 http://mongodb.org/downloads
12.2.3 Release Notes for MongoDB 2.2

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.
- 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 717) 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.
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:

822 https://jira.mongodb.org/security/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.3.2%22,+%222.3.1%22,+%222.3.0%22,+%222.4.0-rc0%22,+%222.4.0-rc1%22,+%222.4.0-rc2%22,+%222.4.0-rc3%22
823 https://jira.mongodb.org/issues/?jql=project%20%3D%20SERVER%20AND%20fixVersion%20in%20%28%222.3.2%22%2C%222.3.1%22%2C%222.3.0%22%2C%222.4.0-rc0%22%2C%222.4.0-rc1%22%2C%222.4.0-rc2%22%2C%222.4.0-rc3%22%22AND%20%22Backwards%20Compatibility%22%20%20%20%20%20%20%22Major%22
824 https://github.com/mongodb/mongo/blob/v2.4/dist/src/THIRD-PARTY-NOTICES
825 http://downloads.mongodb.org/
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 580) 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 717).
- Upgrade all mongos instances *first*, in any order.
- Upgrade all of the mongod config server instances using the *stand alone* (page 900) procedure. To keep the cluster online, be sure that at all times at least one config server is up.
- Upgrade each shard’s replica, using the *upgrade procedure for replica sets* (page 900) detailed above.
- re-enable the balancer.

**Note:** Balancing is not currently supported in mixed 2.0.x and 2.2.0 deployments. Thus you will want to reach a consistent version for all shards within a reasonable period of time, e.g. same-day. See SERVER-6902 for more information.

### 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 469)
- Examples: *Aggregation Examples* (page 452)

[^52]: [https://jira.mongodb.org/browse/SERVER-6902](https://jira.mongodb.org/browse/SERVER-6902)
**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 211) tutorial.

**Concurrency Improvements**  MongoDB 2.2 increases the server’s capacity for concurrent operations with the following improvements:

1. DB Level Locking\(^{827}\)
2. Improved Yielding on Page Faults\(^{828}\)
3. Improved Page Fault Detection on Windows\(^{829}\)

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use. See *locks*, *recordStats*\(^{830}\), *db.currentOp()*\(^{831}\)*, 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.

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* (page 588) and *write concern* (page 78). 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 682) and *Manage Shard Tags* (page 730).

**Fully Supported Read Preference Semantics**  All MongoDB clients and drivers now support full *read preferences* (page 588), including consistent support for a full range of *read preference modes* (page 657) and *tag sets* (page 590). 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.

\(^{827}\)https://jira.mongodb.org/browse/SERVER-4328  
\(^{828}\)https://jira.mongodb.org/browse/SERVER-3357  
\(^{829}\)https://jira.mongodb.org/browse/SERVER-4538  
\(^{830}\)http://docs.mongodb.org/v2.2/reference/server-status
In sharded environments, to ensure that your cluster remains available during the upgrade process you **must** use the upgrade procedure for sharded clusters (page 901).

**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\(^{831}\) 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\(^{832}\) for more information.

**ObjectId().toString() Returns String Literal ObjectId("...")**  In version 2.2, the `toString()` method returns the string representation of the `ObjectId()` (page 176) object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` method on the `ObjectId("507c7f79b0cf7994f6c0e")` object:

```
ObjectId("507c7f79b0cf7994f6c0e").toString()
```

The method now returns the *string* `ObjectId("507c7f79b0cf7994f6c0e")`.

Previously, in version 2.0, the method would return the *hexadecimal string* `507c7f79b0cf7994f6c0e`.

If compatibility between versions 2.0 and 2.2 is required, use `ObjectId().str` (page 176), 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 176) object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` method on the `ObjectId("507c7f79b0cf7994f6c0e")` object:

```
ObjectId("507c7f79b0cf7994f6c0e").valueOf()
```

The method now returns the *hexadecimal string* `507c7f79b0cf7994f6c0e`.

Previously, in version 2.0, the method would return the *object* `ObjectId("507c7f79b0cf7994f6c0e")`.

If compatibility between versions 2.0 and 2.2 is required, use `ObjectId().str` (page 176) attribute, which holds the hexadecimal string value in both versions.

\(^{831}\)https://jira.mongodb.org/browse/SERVER-6226

\(^{832}\)https://jira.mongodb.org/browse/SERVER-6961

12.2. Previous Stable Releases 903
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\(^833\) and the Are there any restrictions on the names of Collections? (page 753) FAQ item.

Restrictions on Database Names for Windows  Database names running on Windows can no longer contain the following characters:

/\.*<>:|?

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, \mongodb will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584\(^834\) and SERVER-6729\(^835\) 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\(^836\) 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\(^837\) and SERVER-828\(^838\) 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\(^839\) for more information.

\(^833\)https://jira.mongodb.org/browse/SERVER-4442
\(^834\)https://jira.mongodb.org/browse/SERVER-4584
\(^835\)https://jira.mongodb.org/browse/SERVER-6729
\(^836\)https://jira.mongodb.org/browse/SERVER-5516
\(^837\)https://jira.mongodb.org/browse/SERVER-2238
\(^838\)https://jira.mongodb.org/browse/SERVER-828
\(^839\)https://jira.mongodb.org/browse/SERVER-5648
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 30) and SERVER-1589840 for more information.

Log Rotate Command Support  MongoDB for Windows now supports log rotation by way of the `logRotate` database command. See SERVER-2612841 for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency  Labeled “2008+” on the Downloads Page842, 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-3844843 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 you attempt to restore this backup with `mongorestore`, the target `mongod` will rebuild all indexes. See SERVER-808844 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-3873845 case and the `mongooplog` reference.

Authentication Support for `mongotop` and `mongostat`  `mongotop` and `mongostat` now contain support for username/password authentication. See SERVER-3875846 and SERVER-3871847 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-3937848 for more information.

In `mongorestore`, the `--w` option provides support for configurable write concern.

---

840 https://jira.mongodb.org/browse/SERVER-1589
841 https://jira.mongodb.org/browse/SERVER-2612
842 http://www.mongodb.org/downloads
843 https://jira.mongodb.org/browse/SERVER-3844
844 https://jira.mongodb.org/browse/SERVER-808
845 https://jira.mongodb.org/browse/SERVER-3873
846 https://jira.mongodb.org/browse/SERVER-3875
847 https://jira.mongodb.org/browse/SERVER-3871
848 https://jira.mongodb.org/browse/SERVER-3937
mongodump Support for Reading from Secondaries  
You can now run mongodump when connected to a secondary member of a replica set. See SERVER-3854 for more information.

mongoimport Support for full 16MB Documents  
Previously, mongoimport would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use mongoimport to import documents that are at least 16 megabytes in size. See SERVER-4593 for more information.

Timestamp() Extended JSON format  
MongoDB extended JSON now includes a new Timestamp() type to represent the Timestamp type that MongoDB uses for timestamps in the oplog among other contexts.

This permits tools like mongooplog and mongodump to query for specific timestamps. Consider the following mongodump operation:

```
mongodump --db local --collection oplog.rs --query '{"ts": "$gt": "$timestamp" : {"t": 1344969612000, "i": 1 }}' --out oplog-dump
```

See SERVER-3483 for more information.

Shell Improvements

Improved Shell User Interface  
2.2 includes a number of changes that improve the overall quality and consistency of the user interface for the mongo shell:

- Full Unicode support.
- Bash-like line editing features. See SERVER-4312 for more information.
- Multi-line command support in shell history. See SERVER-3470 for more information.
- Windows support for the edit command. See SERVER-3998 for more information.

Helper to load Server-Side Functions  
The db.loadServerScripts() loads the contents of the current database’s system.js collection into the current mongo shell session. See SERVER-1651 for more information.

Support for Bulk Inserts  
If you pass an array of documents to the insert() method, the mongo shell will now perform a bulk insert operation. See SERVER-3819 and SERVER-2395 for more information.

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 case and the documentation of the syslogFacility run-time option or the mongod --syslog and mongos --syslog command line-options.

https://jira.mongodb.org/browse/SERVER-3854
https://jira.mongodb.org/browse/SERVER-4593
https://jira.mongodb.org/browse/SERVER-3483
https://jira.mongodb.org/browse/SERVER-4312
https://jira.mongodb.org/browse/SERVER-3470
https://jira.mongodb.org/browse/SERVER-3998
https://jira.mongodb.org/browse/SERVER-1651
https://jira.mongodb.org/browse/SERVER-3819
https://jira.mongodb.org/browse/SERVER-2395
https://jira.mongodb.org/browse/SERVER-2957
**touch Command**  Added the `touch` command to read the data and/or indexes from a collection into memory. See SERVER-2023\(^{859}\) 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\(^{860}\) and indexCounters for more information.

**Padding Specifiable on compact Command**  See the documentation of the `compact` and the SERVER-4018\(^{861}\) issue for more information.

**Added Build Flag to Use System Libraries**  The Boost library, version 1.49, is now embedded in the MongoDB code base.

If you want to build MongoDB binaries using system Boost libraries, you can pass `scons` using the `--use-system-boost` flag, as follows:

```
scons --use-system-boost
```

When building MongoDB, you can also pass `scons` a flag to compile MongoDB using only system libraries rather than the included versions of the libraries. For example:

```
scons --use-system-all
```

See the SERVER-3829\(^{862}\) and SERVER-5172\(^{863}\) 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\(^{864}\) and SERVER-4683\(^{865}\). For more information about TCMalloc, see the documentation of TCMalloc\(^{866}\) 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\(^{867}\) for more information.

**Replica Set Members can Sync from Specific Members**  The new `replSetSyncFrom()` 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.

\(^{859}\)https://jira.mongodb.org/browse/SERVER-2023
\(^{860}\)https://jira.mongodb.org/browse/SERVER-5784
\(^{861}\)https://jira.mongodb.org/browse/SERVER-4018
\(^{862}\)https://jira.mongodb.org/browse/SERVER-3829
\(^{863}\)https://jira.mongodb.org/browse/SERVER-5172
\(^{864}\)https://jira.mongodb.org/browse/SERVER-188
\(^{865}\)https://jira.mongodb.org/browse/SERVER-4683
\(^{866}\)http://goog-perftools.sourceforge.net/doc/tcmalloc.html
\(^{867}\)https://jira.mongodb.org/browse/SERVER-3575
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` set to `true`, other members of the replica set will `not` sync from this member, unless they also have `buildIndexes` set to `true`. See SERVER-4160 for more information.

**New Option To Configure Index Pre-Fetching during Replication**  By default, when replicating options, `secondaries` will pre-fetch `Indexes` (page 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 for more information.

**Map Reduce Improvements**

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce, and
- MapReduce will retry jobs following a config error.

**Sharding Improvements**

**Index on Shard Keys Can Now Be a Compound Index**  If your shard key uses the prefix of an existing index, then you do not need to maintain a separate index for your shard key in addition to your existing index. This index, however, cannot be a multi-key index. See the _Shard Key Indexes_ (page 688) documentation and SERVER-1506 for more information.

**Migration Thresholds Modified**  The _migration thresholds_ (page 685) 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 685) documentation for more information.

**Licensing Changes**

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice and the SERVER-4683 for more information.

**Resources**

- MongoDB Downloads.
- All JIRA issues resolved in 2.2.
- All backwards incompatible changes.
• All third party license notices\(^{878}\).
• What’s New in MongoDB 2.2 Online Conference\(^{879}\).

## 12.2.4 Release Notes for MongoDB 2.0

### 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\(^{880}\).

**Journaling** (page 308) 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]\(^{881}\).
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]\(^{882}\).
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 78) 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 580) procedure.

`rs.stepDown()` and `replSetStepDown` provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

---

\(^{878}\)https://github.com/mongodb/mongo/blob/v2.2/dist/src/THIRD-PARTY-NOTICES

\(^{879}\)http://www.mongodb.com/events/webinar/mongodb-online-conference-sept

\(^{880}\)https://jira.mongodb.org/browse/SERVER-1097

\(^{881}\)http://downloads.mongodb.org/

\(^{882}\)http://downloads.mongodb.org/

---

12.2. Previous Stable Releases 909
When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` binary with the 2.0.x binary.

**Upgrading a Sharded Cluster**

1. Upgrade all `config server` instances *first*, in any order. Since config servers use two-phase commit, *shard* configuration metadata updates will halt until all are up and running.
2. Upgrade `mongos` 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<sup>883</sup> 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 523). 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<sup>884</sup>.

<sup>883</sup>https://jira.mongodb.org/browse/SERVER-2563
<sup>884</sup>https://jira.mongodb.org/browse/SERVER-3866
• The `repairDatabase` command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the 2.0 type (page 910), 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 523).

**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* 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 78) 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:

```javascript
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 207).

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 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 78).

---

12.2. Previous Stable Releases
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 638).

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 631).

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 499).

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 274).

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[^886]
- All JIRA Issues resolved in 2.0[^887]
- All Backward Incompatible Changes[^888]

### 12.2.5 Release Notes for MongoDB 1.8

**Upgrading**

MongoDB 1.8 is a standard, incremental production release and works as a drop-in replacement for MongoDB 1.6, except:

- *Replca set* members should be upgraded in a particular order, as described in *Upgrading a Replica Set* (page 915).
- 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[^889].
2. Shutdown your `mongod` instance.
3. Replace the existing binary with the 1.8.x `mongod` binary.
4. Restart MongoDB.

[^886]: http://mongodb.org/downloads
[^887]: https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002
[^889]: 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.\(^{890}\)

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.config()
   {
   "_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",
   \(^{890}\)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\(^\text{891}\).

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\(^\text{892}\).

**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 915).
   - 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\(^\text{893}\).

3. For each mongos:
   (a) Shut down the mongos process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page\(^\text{894}\).
4. For each config server:
   (a) Shut down the config server process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page.
5. Turn on the balancer:

   ```
   use config
   db.settings.update({_id: "balancer"}, {$set: {stopped: false}})
   ```

### Returning to 1.6

If for any reason you must move back to 1.6, follow the steps above in reverse. Please be careful that you have not inserted any documents larger than 4MB while running on 1.8 (where the max size has increased to 16MB). If you have you will get errors when the server tries to read those documents.

**Journaling** Returning to 1.6 after using 1.8 *Journaling* (page 308) 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 308) 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 505) 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 66) 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:

---

895 [http://downloads.mongodb.org/](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 308).

• 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 505).

1.7.3

• Initial covered index (page 66) 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 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\textsuperscript{896}, 1.8.0\textsuperscript{897}
• 1.7.6\textsuperscript{898}, 1.7.5\textsuperscript{899}, 1.7.4\textsuperscript{900}, 1.7.3\textsuperscript{901}, 1.7.2\textsuperscript{902}, 1.7.1\textsuperscript{903}, 1.7.0\textsuperscript{904}

\textsuperscript{896}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/v09MbhEm62Y
\textsuperscript{897}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/JeHQOnam6Qk
\textsuperscript{898}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/3t6GNZ1qGyc
\textsuperscript{899}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/S5R0Tx9wkEg
\textsuperscript{900}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/9Om3Vuw-y9c
\textsuperscript{901}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/DrNUrdbmIi
\textsuperscript{902}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/dT7nwK6XiSo
\textsuperscript{903}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/HUR9zYtPAP4
\textsuperscript{904}https://groups.google.com/forum/?fromgroups#!topic/mongodb-user/TUnJCg9161A

12.2. Previous Stable Releases
12.2.6 Release Notes for MongoDB 1.6

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 661) 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 661)
- Deploy a Sharded Cluster (page 691)
- Convert a Replica Set to a Replicated Sharded Cluster (page 699)

Replica Sets

Replica sets (page 559), 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 559)
- Deploy a Replica Set (page 603)
- Convert a Standalone to a Replica Set (page 614)

Other Improvements

- The \texttt{w} option (and \texttt{wtimeout}) forces writes to be propagated to \texttt{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 \texttt{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

1.5.x Release Notes

• 1.5.8
• 1.5.7
• 1.5.6
• 1.5.5
• 1.5.4
• 1.5.3
• 1.5.2
• 1.5.1
• 1.5.0

You can see a full list of all changes on JIRA.

Thank you everyone for your support and suggestions!

12.2.7 Release Notes for MongoDB 1.4

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, in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/06_QCC05Fpk
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/uJfF1QN6Thk
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/OYvz40RWs90
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/4l0N2U_H0cQ
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/oO749nvTARY
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/380V_Ec_q1c
https://groups.google.com/forum/?hl=en&fromgroups=#!topic/mongodb-user/hsUQL9CxTQw
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/94EE3HVidAA
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/5SBPQ2RsfdM
https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/VAhkcJDG5y0
https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107
Core Server Enhancements

- concurrency (page 758) improvements
- indexing memory improvements
- background index creation (page 508)
- 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 228)
- ability to do fsync + lock for backing up raw files
- option for separate directory per db (--directoryperdb)
- http://localhost:28017/_status to get serverStatus via http
- REST interface is off by default for security (--rest to enable)
- can rotate logs with a db command, logRotate
- enhancements to serverStatus command (db.serverStatus()) - counters and replication lag (page 647) 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 499)
- geo $center and $box searches

12.2.8 Release Notes for MongoDB 1.2.x

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

- mongoimport json has been removed and is replaced with mongoimport that can do json/csv/tsv

field filter changing

- We’ve changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter only changes the output, not which objects are returned. If you need that behavior, you can use $exists
12.3 Other MongoDB Release Notes

12.3.1 Default Write Concern Change

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.
   
   This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.
   
   The new top-level connection class is named MongoClient, or similar depending on how host languages handle namespaces.

2. The default write concern on the new MongoClient class will be to acknowledge all write operations. This will allow your application to receive acknowledgment of all write operations.
   
   See the documentation of Write Concern (page 78) 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 924). 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.

918 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 setting in the Replica Set Configuration (page 652).
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

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

The MongoDB Manual contains comprehensive documentation on the MongoDB document-oriented database management system. This page describes the manual’s licensing, editions, and versions, and describes how to make a change request and how to contribute to the manual.

For more information on MongoDB, see MongoDB: A Document Oriented Database. To download MongoDB, see the downloads page.

13.1 License

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


13.2 Editions

In addition to the MongoDB Manual, you can also access this content in the following editions:

- ePub Format
- Single HTML Page
- PDF Format (without reference.)
- HTML tar.gz

You also can access PDF files that contain subsets of the MongoDB Manual:

- MongoDB Reference Manual
- MongoDB CRUD Operations

1http://docs.mongodb.org/manual/
2http://www.mongodb.org/about/
3http://www.mongodb.org/downloads
4http://creativecommons.org/licenses/by-nc-sa/3.0/
5http://docs.mongodb.org/manual/
6http://docs.mongodb.org/master/MongoDB-manual.epub
7http://docs.mongodb.org/master/single/
8http://docs.mongodb.org/master/MongoDB-manual.pdf
9http://docs.mongodb.org/master/manual.tar.gz
11http://docs.mongodb.org/master/MongoDB-crud-guide.pdf
• Data Models for MongoDB
• MongoDB Data Aggregation
• Replication and MongoDB
• Sharding and MongoDB
• MongoDB Administration
• MongoDB Security

MongoDB Reference documentation is also available as part of dash. You can also access the MongoDB Man Pages which are also distributed with the official MongoDB Packages.

13.3 Version and Revisions

This version of the manual reflects version 3.0 of MongoDB.

See the MongoDB Documentation Project Page for an overview of all editions and output formats of the MongoDB Manual. You can see the full revision history and track ongoing improvements and additions for all versions of the manual from its GitHub repository.

This edition reflects “master” branch of the documentation as of the “ec7c26956613981e2632b9108f0677304b1d09c1” revision. This branch is explicitly accessible via “http://docs.mongodb.org/master” and you can always reference the commit of the current manual in the release.txt file.

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

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.

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.
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 Agreement²⁴, and
- join the mongodb-translators²⁵ user group.

The mongodb-translators²⁶ 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 928)
- Style Guide and Documentation Conventions (page 930)
- MongoDB Manual Organization (page 939)
- MongoDB Documentation Practices and Processes (page 936)
- MongoDB Documentation Build System (page 940)

The entire documentation source for this manual is available in the mongodb/docs repository²⁷, which is one of the MongoDB project repositories on GitHub²⁸.

To contribute to the documentation, you can open a GitHub account²⁹, fork the mongodb/docs repository³⁰, make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the MongoDB Contributor Agreement³¹.

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 Sphinx³², a sophisticated documentation engine built upon Python Docutils³³. The original reStructured Text³⁴ 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:

²⁴http://www.mongodb.com/legal/contributor-agreement
²⁵http://groups.google.com/group/mongodb-translators
²⁶http://groups.google.com/group/mongodb-translators
²⁷https://github.com/mongodb/docs
²⁸http://github.com/mongodb
²⁹https://github.com/
³⁰https://github.com/mongodb/docs
³¹http://www.mongodb.com/contributor
³²http://sphinx-doc.org/
³³http://docutils.sourceforge.net/
³⁴http://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 939).

Document History

2011-09-27: Document created with a (very) rough list of style guidelines, conventions, and questions.
2012-01-12: Document revised based on slight shifts in practice, and as part of an effort of making it easier for people outside of the documentation team to contribute to documentation.
2012-03-21: Merged in content from the Jargon, and cleaned up style in light of recent experiences.
2012-08-10: Addition to the “Referencing” section.
2013-02-07: Migrated this document to the manual. Added “map-reduce” terminology convention. Other edits.

Naming Conventions

This section contains guidelines on naming files, sections, documents and other document elements.

- File naming Convention:
  - For Sphinx, all files should have a .txt extension.
  - Separate words in file names with hyphens (i.e. -).
  - For most documents, file names should have a terse one or two word name that describes the material covered in the document. Allow the path of the file within the document tree to add some of the required context/categorization. For example it’s acceptable to have http://docs.mongodb.org/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-shard-cluster.rst.

- 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 699) 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 699).

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.
<table>
<thead>
<tr>
<th>Preferred Term</th>
<th>Concept</th>
<th>Dispreferred Alternatives</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
<td>A single, top-level object/record in a MongoDB collection.</td>
<td>record, object, row</td>
<td>Prefer document over object because of concerns about cross-driver language handling of objects. Reserve record for “allocation” of storage. Avoid “row,” as possible.</td>
</tr>
<tr>
<td>database</td>
<td>A group of collections. Refers to a group of data files. This is the “logical” sense of the term “database.”</td>
<td></td>
<td>Avoid “primary” construction. Typically in the form MongoDB deployment.</td>
</tr>
<tr>
<td>instance</td>
<td>A daemon process. (e.g. mongos or mongod)</td>
<td>process (acceptable sometimes), node (never acceptable), server.</td>
<td>Avoid confusion with the command, shell helper, and driver interfaces. Makes it possible to discuss the operation generally. 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.</td>
</tr>
<tr>
<td>field name</td>
<td>The identifier of a value in a document.</td>
<td>key, column</td>
<td></td>
</tr>
<tr>
<td>field/value</td>
<td>The name/value pair that describes a unit of data in MongoDB.</td>
<td>key, slot, attribute</td>
<td>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.</td>
</tr>
<tr>
<td>MongoDB value</td>
<td>The data content of a field. A group of processes, or deployment that implement the MongoDB interface.</td>
<td>data mongo, mongod, cluster</td>
<td>Stylistic preference, mostly. In some cases it’s useful to be able to refer generically to instances (that may be either mongod or mongos.)</td>
</tr>
<tr>
<td>embedded</td>
<td>An embedded or nested document within a document or an array.</td>
<td>embedded document, nested document</td>
<td></td>
</tr>
<tr>
<td>document map</td>
<td>An operation performed by the mapReduce command.</td>
<td>mapReduce, map reduce, map/reduce, grid, shard cluster, set, deployment</td>
<td></td>
</tr>
<tr>
<td>cluster</td>
<td>A sharded cluster.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sharded cluster</td>
<td>A sharded cluster.</td>
<td>shard cluster, cluster, sharded system set, replication deployment</td>
<td></td>
</tr>
<tr>
<td>replica set</td>
<td>A deployment of replicating mongod programs that provide redundancy and automatic failover.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deployment data</td>
<td>A group of MongoDB processes, or a standalone mongod instance. The collection of physical mongod instances (that may be either mongod or mongos.)</td>
<td>cluster, system database, data</td>
<td>Important to keep the distinction between the operation generally. 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.</td>
</tr>
<tr>
<td>mongos</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 matters, 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-4135.)

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.org36. Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

35https://jira.mongodb.org/browse/DOCS-41
36http://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 proposing the change. Someone on the documentation team will make the change and be in contact with you so that you can review the change.
- Using GitHub, fork the mongodb/docs repository, commit your changes, and issue a pull request. Someone on the documentation team will review and incorporate your change into the documentation.

** Builds**

Building the documentation is useful because Sphinx and docutils can catch numerous errors in the format and syntax of the documentation. Additionally, having access to an example documentation as it will appear to the users is useful for providing more effective basis for the review process. Besides Sphinx, Pygments, and Python-Docutils, the documentation repository contains all requirements for building the documentation resource.

Talk to someone on the documentation team if you are having problems running builds yourself.

**Publication**

The makefile for this repository contains targets that automate the publication process. Use make html to publish a test build of the documentation in the build/ directory of your repository. Use make publish to build the full contents of the manual from the current branch in the ../public-docs/ directory relative the docs repository.

Other targets include:

- 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.

---

37[https://jira.mongodb.org/browse/DOCS](https://jira.mongodb.org/browse/DOCS)
38[https://github.com/](https://github.com/)
39[https://github.com/mongodb/docs](https://github.com/mongodb/docs)
40[http://sphinx.pocoo.org/](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\(^{41}\) 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\(^{42}\) or contact the documentation team\(^{43}\).

---

\(^{41}\)http://jira.mongodb.org/browse/DOCS
\(^{42}\)https://jira.mongodb.org/browse/DOCS
\(^{43}\)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\[44\] structure. However, in order to annotate the table of contents and provide additional flexibility, the MongoDB documentation generates toctree\[45\] 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\[46\] 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\[47\] 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

**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.

**Optional**

To generate PDFs for the full production build, install a TeX distribution (for building the PDF.) If you do not have a LaTeX installation, use MacTeX. This is **only** required to build PDFs.

**Arch Linux**  Install packages from the system repositories with the following command:

```
pacman -S 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
```

---

48 [https://pypi.python.org/pypi/giza](https://pypi.python.org/pypi/giza)
49 [http://inkscape.org/download/](http://inkscape.org/download/)
Setup and Configuration  Clone the repository:

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

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:

```
made 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\(^{51}\).

---

\(^{51}\)https://jira.mongodb.org/browse/DOCS