MongoDB Documentation

Release 2.2.7

MongoDB Documentation Project

April 03, 2015

Contents

See *About MongoDB Documentation* (page 423) for more information about the MongoDB Documentation project, this Manual and additional editions of this text.

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

[1] http://docs.mongodb.org/v2.2/MongoDB-reference-manual.pdf

CHAPTER 1

Installing MongoDB

1.1 Installation Guides

MongoDB runs on most platforms and supports 32-bit and 64-bit architectures. MongoDB is available as a binary, or as a package. In production environments, use 64-bit MongoDB binaries. Choose your platform below:

1.1.1 Install MongoDB on Red Hat Enterprise, CentOS, or Fedora

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

Packages

The MongoDB downloads repository contains two packages:

- `mongo-10gen-server`

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

- `mongo-10gen`

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

Install MongoDB

Configure Package Management System (YUM)

Create a `/etc/yum.repos.d/mongodb.repo` file to hold the following configuration information for the MongoDB repository:

Tip

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

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

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

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

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

Install Packages

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

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

When this command completes, you have successfully installed MongoDB!

Manage Installed Versions

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

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

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

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

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

Control Scripts

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

[a] https://jira.mongodb.org/browse/SERVER-7285

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

As of version 2.2.7, there are no control scripts for `mongos`. `mongos` is only used in *sharding deployments* (page 295). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB

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

- enable access to the relevant ports (e.g. 27017) for SELinux. See *Interfaces and Port Numbers* (page 134) for more information on MongoDB's default ports.
- disable SELinux entirely. This requires a system reboot and may have larger implications for your deployment.

Start MongoDB

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

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

```
service mongod start
```

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

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

```
chkconfig mongod on
```

Stop MongoDB

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

```
service mongod stop
```

Restart MongoDB

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

```
service mongod restart
```

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

1.1.2 Install MongoDB on Ubuntu

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

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

Package Options

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

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

Install MongoDB

Configure Package Management System (APT)

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

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

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

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

Now issue the following command to reload your repository:

```
sudo apt-get update
```

Install Packages

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

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

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

Manage Installed Versions

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

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

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

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

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

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

Control Scripts

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

As of version 2.2.7, there are no control scripts for `mongos`. `mongos` is only used in *sharding deployments* (page 295). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB

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

Start MongoDB

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

```
sudo service mongodb start
```

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

Stop MongoDB

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

```
sudo service mongodb stop
```

Restart MongoDB

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

```
sudo service mongodb restart
```

1.1.3 Install MongoDB on Debian

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

Note: This tutorial applies to both Debian systems and versions of Ubuntu Linux prior to 9.10 "Karmic" which do not use Upstart. Other Ubuntu users will want to follow the *Install MongoDB on Ubuntu* (page 5) tutorial.

Package Options

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

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

Install MongoDB

Configure Package Management System (APT)

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

Step 1: Import MongoDB PGP Key

Issue the following command to add the MongoDB public GPG Key[2] to the system key ring.

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

Step 2: Create a `sources.list` file for MongoDB

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

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

Step 3: Reload Local Package Database

Issue the following command to reload the local package database:

```
sudo apt-get update
```

Install Packages

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

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

When this command completes, you have successfully installed MongoDB!

Manage Installed Versions

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

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

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

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

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

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

Control Scripts

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

As of version 2.2.7, there are no control scripts for `mongos`. `mongos` is only used in *sharding deployments* (page 295). You can use the `mongod` init script to derive your own `mongos` control script.

Run MongoDB

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

Start MongoDB

Issue the following command to start `mongod`:

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

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

Stop MongoDB

Issue the following command to stop `mongod`:

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

Restart MongoDB

Issue the following command to restart `mongod`:

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

1.1.4 Install MongoDB on Linux Systems

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

Installation Process

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

Install for 64-bit Linux

Step 1: Download the Latest Release In a system shell, download the latest release for 64-bit Linux.

```
curl -O http://downloads.mongodb.org/linux/mongodb-linux-x86_64-2.2.7.tgz
```

You may optionally specify a different version to download.

Step 2: Extract MongoDB From Archive Extract the files from the downloaded archive.

```
tar -zxvf mongodb-linux-x86_64-2.2.7.tgz
```

Step 3: Optional: Copy MongoDB to Target Directory Copy the extracted folder into another location, such as `mongodb`.

```
mkdir -p mongodb
cp -R -n mongodb-linux-x86_64-2.2.7/ mongodb
```

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

Install for 32-bit Linux

Step 1: Download the Latest Release In a system shell, download the latest release for 32-bit Linux.

```
curl -O http://downloads.mongodb.org/linux/mongodb-linux-i686-2.2.7.tgz
```

You may optionally specify a different version to download.

Step 2: Extract MongoDB From Archive Extract the files from the downloaded archive.

```
tar -zxvf mongodb-linux-i686-2.2.7.tgz
```

Step 3: Optional: Copy MongoDB to Target Directory Copy the extracted folder into another location, such as `mongodb`.

```
mkdir -p mongodb
cp -R -n mongodb-linux-i686-2.2.7/ mongodb
```

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

Run MongoDB

Set Up the Data Directory

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

Step 1: Create `dbpath` To create the default `dbpath` directory, use the following command:

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

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

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

You must create the `mongodb` user separately.

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

Start MongoDB

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

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

```
mongod
```

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

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

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

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

Stop MongoDB

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

1.1.5 Install MongoDB on OS X

Platform Support

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

MongoDB is available through the popular OS X package manager Homebrew[3] or through the MongoDB Download site.

[3] http://brew.sh/

Install MongoDB with Homebrew

Homebrew[4] [5] installs binary packages based on published "formulae". The following commands will update `brew` to the latest packages and install MongoDB.

In a terminal shell, use the following sequence of commands to update‘‘brew‘‘ to the latest packages and install MongoDB:

```
brew update
brew install mongodb
```

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

```
brew update
brew upgrade mongodb
```

Optionally, you can choose to build MongoDB from source. Use the following command to build MongoDB with SSL support:

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

You can also install the latest development release of MongoDB for testing and developemnt with the following command:

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

Manual Installation

Step 1: Download the Latest Release

In a system shell, download the latest release for 64-bit OS X.

```
curl -O http://downloads.mongodb.org/osx/mongodb-osx-x86_64-2.2.7.tgz
```

You may optionally specify a different version to download.

Step 2: Extract MongoDB From Archive

Extract the files from the downloaded archive.

```
tar -zxvf mongodb-osx-x86_64-2.2.7.tgz
```

Step 3: Optional: Copy MongoDB to Target Directory

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

```
mkdir -p mongodb
cp -R -n mongodb-osx-x86_64-2.2.7/ mongodb
```

[4] http://brew.sh/

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

Step 4: Optional: Configure Search Path

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

Run MongoDB

Set Up the Data Directory

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

Step 1: Create `dbpath` To create the default `dbpath` directory, use the following command:

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

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

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

You must create the `mongodb` user separately.

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

Start MongoDB

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

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

```
mongod
```

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

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

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

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

Stop MongoDB

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

1.1.6 Install MongoDB on Windows

Synopsis

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

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

Procedure

Important: If you are running any edition of Windows Server 2008 R2 or Windows 7, please install a hotfix to resolve an issue with memory mapped files on Windows[6].

Download MongoDB for Windows

Download the latest production release of MongoDB from the MongoDB downloads page[7].

There are three builds of MongoDB for Windows:

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

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

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

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

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

```
wmic os get osarchitecture
```

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

Note: The folder name will be either:

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

[6] http://support.microsoft.com/kb/2731284
[7] http://www.mongodb.org/downloads

Or:

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

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

Set up the Environment

Start the *Command Prompt* by selecting the *Start Menu*, then *All Programs*, then *Accessories*, then right click *Command Prompt*, and select *Run as Administrator* from the popup menu. In the *Command Prompt*, issue the following commands:

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

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

MongoDB requires a *data folder* to store its files. The default location for the MongoDB data directory is `C:\data\db`. Create this folder using the *Command Prompt*. Issue the following command sequence:

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

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

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

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

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

Start MongoDB

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

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

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

Note: Depending on the security level of your system, Windows will issue a *Security Alert* dialog box about blocking "some features" of `C:\\mongodb\bin\mongod.exe` from communicating on networks. All users should select `Private Networks, such as my home or work network` and click `Allow access`. For additional information on security and MongoDB, please read the *Security Practices and Management* (page 133) page.

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

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

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

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

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

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

See also:

"`mongo`" and "`http://docs.mongodb.org/manual/reference/javascript`." If you want to develop applications using .NET, see the documentation of C# and MongoDB[8] for more information.

MongoDB as a Windows Service

New in version 2.0.

Setup MongoDB as a *Windows Service*, so that the database will start automatically following each reboot cycle.

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

Configure the System

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

1. Create a specific directory for MongoDB log files:

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

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

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

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

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

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

[8] http://docs.mongodb.org/ecosystem/drivers/csharp

Install and Run the MongoDB Service

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

1. To install the MongoDB service:

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

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

2. To run the MongoDB service:

   ```
   net start MongoDB
   ```

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

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

Stop or Remove the MongoDB Service

- To stop the MongoDB service:

  ```
  net stop MongoDB
  ```

- To remove the MongoDB service:

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

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

1.1.7 Getting Started with MongoDB Development

This tutorial provides an introduction to basic database operations using the `mongo` shell. `mongo` is a part of the standard MongoDB distribution and provides a full JavaScript environment with a complete access to the JavaScript language and all standard functions as well as a full database interface for MongoDB. See the mongo JavaScript API[9] documentation and the `mongo` shell `JavaScript Method Reference`.

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

[9] http://api.mongodb.org/js

This tutorial addresses the following aspects of MongoDB use:

Connect to a Database

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

Connect to a `mongod`

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

```
mongo
```

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

Select a Database

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

```
db
```

`db` returns the name of the current database.

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

   ```
   show dbs
   ```

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

   ```
   use mydb
   ```

3. Confirm that your session has the `mydb` database as context, using the `db` operation, which returns the name of the current database as follows:

   ```
   db
   ```

At this point, if you issue the `show dbs` operation again, it will not include `mydb`, because MongoDB will not create a database until you insert data into that database. The *Create a Collection and Insert Documents* (page 19) section describes the process for inserting data.

Display `mongo` Help

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

```
help
```

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

Create a Collection and Insert Documents

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

MongoDB will create collections and databases implicitly upon their first use: you do not need to create the database or collection before inserting data. Furthermore, because MongoDB uses *dynamic schemas* (page 356), you do not need to specify the structure of your documents before inserting them into the collection.

Insert Individual Documents

1. From the `mongo` shell, confirm that the current context is the `mydb` database with the following operation:

   ```
   db
   ```

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

   ```
   use mydb
   ```

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

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

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

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

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

5. Confirm that the collection named `things` exists using the following operation:

   ```
   show collections
   ```

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

6. Confirm that the documents exist in the collection `things` by issuing query on the `things` collection. Using the `find()` method in an operation that resembles the following:

   ```
   db.things.find()
   ```

   This operation returns the following results. The *ObjectId* (page 54) values will be unique:

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

All MongoDB documents must have an `_id` field with a unique value. These operations do not explicitly specify a value for the `_id` field, so `mongo` creates a unique *ObjectId* (page 54) value for the field before inserting it into the collection.

Insert Multiple Documents Using a For Loop

1. From the `mongo` shell, add more documents to the `things` collection using the following `for` loop:

   ```
   for (var i = 1; i <= 20; i++) db.things.insert( { x : 4 , j : i } )
   ```

2. Query the collection by issuing the following command:

   ```
   db.things.find()
   ```

   The `mongo` shell displays the first 20 documents in the collection. Your *ObjectId* (page 54) values will be different:

   ```
   { "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
   { "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
   { "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
   { "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
   { "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
   { "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
   { "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
   { "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
   { "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
   { "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
   { "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
   { "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
   ```

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

   ```
   { "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
   { "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
   ```

For more information on inserting new documents, see the *insert()* (page 62) documentation.

Working with the Cursor

When you query a *collection*, MongoDB returns a "cursor" object that contains the results of the query. The `mongo` shell then iterates over the cursor to display the results. Rather than returning all results at once, the shell iterates over the cursor 20 times to display the first 20 results and then waits for a request to iterate over the remaining results. This prevents `mongo` from displaying thousands or millions of results at once.

The `it` operation allows you to iterate over the next 20 results in the shell. In the *previous procedure* (page 20), the cursor only contained two more documents, and so only two more documents displayed.

The procedures in this section show other ways to work with a cursor. For comprehensive documentation on cursors, see *Iterate the Returned Cursor* (page 74).

Iterate over the Cursor with a Loop

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

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

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

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

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

   The result of this operation follows, although if the *ObjectId* (page 54) values will be unique:

   ```
   { "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
   { "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
   { "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
   { "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
   { "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
   { "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
   { "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
   { "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
   { "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
   { "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
   { "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
   { "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
   { "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
   { "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
   ```

Use Array Operations with the Cursor

You can manipulate a cursor object as if it were an array. Consider the following procedure:

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

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

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

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

   MongoDB returns the following:

```
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
```

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

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

For more information on the cursor, see *Iterate the Returned Cursor* (page 74).

Query for Specific Documents

MongoDB has a rich query system that allows you to select and filter the documents in a collection along specific fields and values. See *Query Document* (page 26) and *Read* (page 69) for a full account of queries in MongoDB.

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

To query for specific documents, do the following:

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

   ```
   db.things.find( { name : "mongo" } )
   ```

   MongoDB returns one document that fits this criteria. The *ObjectId* (page 54) value will be different:

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

2. Query for all documents where `x` has a value of `4` by passing the `{ x : 4 }` query document as a parameter to `find()`:

   ```
   db.things.find( { x : 4 } )
   ```

   MongoDB returns the following result set:

   ```
   { "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
   { "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
   { "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85b"), "x" : 4, "j" : 6 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85c"), "x" : 4, "j" : 7 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85d"), "x" : 4, "j" : 8 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85e"), "x" : 4, "j" : 9 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85f"), "x" : 4, "j" : 10 }
   { "_id" : ObjectId("4c220a42f3924d31102bd860"), "x" : 4, "j" : 11 }
   { "_id" : ObjectId("4c220a42f3924d31102bd861"), "x" : 4, "j" : 12 }
   { "_id" : ObjectId("4c220a42f3924d31102bd862"), "x" : 4, "j" : 13 }
   { "_id" : ObjectId("4c220a42f3924d31102bd863"), "x" : 4, "j" : 14 }
   { "_id" : ObjectId("4c220a42f3924d31102bd864"), "x" : 4, "j" : 15 }
   { "_id" : ObjectId("4c220a42f3924d31102bd865"), "x" : 4, "j" : 16 }
   { "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
   { "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
   { "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
   { "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
   ```

   ObjectId (page 54) values are always unique.

3. Query for all documents where `x` has a value of `4`, as in the previous query, but only return only the value of `j`. MongoDB will also return the `_id` field, unless explicitly excluded. To do this, you add the `{ j : 1 }` document as the *projection* in the second parameter to `find()`. This operation would resemble the following:

   ```
   db.things.find( { x : 4 } , { j : 1 } )
   ```

   MongoDB returns the following results:

   ```
   { "_id" : ObjectId("4c220a42f3924d31102bd856"), "j" : 1 }
   { "_id" : ObjectId("4c220a42f3924d31102bd857"), "j" : 2 }
   { "_id" : ObjectId("4c220a42f3924d31102bd858"), "j" : 3 }
   { "_id" : ObjectId("4c220a42f3924d31102bd859"), "j" : 4 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85a"), "j" : 5 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85b"), "j" : 6 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85c"), "j" : 7 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85d"), "j" : 8 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85e"), "j" : 9 }
   { "_id" : ObjectId("4c220a42f3924d31102bd85f"), "j" : 10 }
   { "_id" : ObjectId("4c220a42f3924d31102bd860"), "j" : 11 }
   { "_id" : ObjectId("4c220a42f3924d31102bd861"), "j" : 12 }
   { "_id" : ObjectId("4c220a42f3924d31102bd862"), "j" : 13 }
   { "_id" : ObjectId("4c220a42f3924d31102bd863"), "j" : 14 }
   { "_id" : ObjectId("4c220a42f3924d31102bd864"), "j" : 15 }
   { "_id" : ObjectId("4c220a42f3924d31102bd865"), "j" : 16 }
   { "_id" : ObjectId("4c220a42f3924d31102bd866"), "j" : 17 }
   { "_id" : ObjectId("4c220a42f3924d31102bd867"), "j" : 18 }
   { "_id" : ObjectId("4c220a42f3924d31102bd868"), "j" : 19 }
   { "_id" : ObjectId("4c220a42f3924d31102bd869"), "j" : 20 }
   ```

Return a Single Document from a Collection

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

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

```
db.things.findOne()
```

For more information on querying for documents, see the *Read* (page 69) and *Read Operations* (page 25) documentation.

Limit the Number of Documents in the Result Set

You can constrain the size of the result set to increase performance by limiting the amount of data your application must receive over the network.

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

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

MongoDB will return the following result, with different *ObjectId* (page 54) values:

```
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
```

Next Steps with MongoDB

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

- *CRUD Operations for MongoDB* (page 61)
- `http://docs.mongodb.org/manual/reference/sql-comparison`
- `http://docs.mongodb.org/manual/applications/drivers`
- *Getting Started with MongoDB Development* (page 17)
- *Create* (page 61)
- *Read* (page 69)
- *Update* (page 77)
- *Delete* (page 83)

1.2 Release Notes

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

- Current Stable Release:
  - *Release Notes for MongoDB 2.2* (page 395)
- Previous Stable Releases:
  - *Release Notes for MongoDB 2.0* (page 404)
  - *Release Notes for MongoDB 1.8* (page 410)

CHAPTER 2

Core MongoDB Operations (CRUD)

CRUD stands for *create*, *read*, *update*, and *delete*, which are the four core database operations used in database driven application development. The *CRUD Operations for MongoDB* (page 61) section provides introduction to each class of operation along with complete examples of each operation. The documents in the *Read and Write Operations in MongoDB* (page 25) section provide a higher level overview of the behavior and available functionality of these operations.

2.1 Read and Write Operations in MongoDB

The *Read Operations* (page 25) and *Write Operations* (page 37) documents provide higher level introductions and description of the behavior and operations of read and write operations for MongoDB deployments. The *BSON Documents* (page 47) provides an overview of *documents* and document-orientation in MongoDB.

2.1.1 Read Operations

Read operations include all operations that return a cursor in response to application request data (i.e. *queries*,) and also include a number of *aggregation* (page 151) operations that do not return a cursor but have similar properties as queries. These commands include `aggregate`, `count`, and `distinct`.

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

Note: All of the examples in this document use the `mongo` shell interface. All of these operations are available in an idiomatic interface for each language by way of the `MongoDB Driver`. See your driver documentation[1] for full API documentation.

Queries in MongoDB

In the `mongo` shell, the `find()` and `findOne()` methods perform read operations. The `find()` method has the following syntax: [2]

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

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

[1] http://api.mongodb.org/

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

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

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

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

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

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

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

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

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

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

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

For additional documentation and examples of the main MongoDB read operators, refer to the *Read* (page 69) page of the *Core MongoDB Operations (CRUD)* (page 25) section.

Query Document

This section provides an overview of the query document for MongoDB queries. See the preceding section for more information on *queries in MongoDB* (page 25).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Result Projections

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

The second argument to the `find()` method is a projection, and it takes the form of a *document* with a list of fields for inclusion or exclusion from the result set. You can either specify the fields to include (e.g. `{ field: 1 }`) or specify the fields to exclude (e.g. `{ field: 0 }`). The `_id` field is implicitly included, unless explicitly excluded.

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

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

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

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

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

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

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

- You can remove the `_id` field by excluding it from the projection, as in the following example:

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

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

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

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

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

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

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

Indexes

Indexes improve the efficiency of read operations by reducing the amount of data that query operations need to process and thereby simplifying the work associated with fulfilling queries within MongoDB. The indexes themselves are a special data structure that MongoDB maintains when inserting or modifying documents, and any given index can: support and optimize specific queries, sort operations, and allow for more efficient storage utilization. For more information about indexes in MongoDB see: *Indexes* (page 185) and *Indexing Overview* (page 185).

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

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

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

  You can create an index on a single field or a *compound index* (page 187) that includes multiple fields in the index.

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

  MongoDB can read the index in either direction. In most cases, you only need to specify *indexing order* (page 188) to support sort operations in compound queries.

Covering a Query

An index *covers* (page 200) a query, a *covered query*, when:

- all the fields in the *query* (page 26) are part of that index, **and**
- all the fields returned in the documents that match the query are in the same index.

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

Example

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

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

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

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

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

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

See *Create Indexes that Support Covered Queries* (page 200) for more information on the behavior and use of covered queries.

Measuring Index Use

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

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

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

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

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

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

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

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

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

Consider the results:

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

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

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

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

  Although the query uses an index to find the matching documents, if `indexOnly` is false then an index could not *cover* (page 30) the query: MongoDB could not both match the *query conditions* (page 26) **and** return the results using only this index. See *Create Indexes that Support Covered Queries* (page 200) for more information.

Query Optimization

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

To create a new query plan, the query optimizer:

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

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

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

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

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

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

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

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

This returns the statistics regarding the execution of the query. For more information on the output of `explain()`, see the `http://docs.mongodb.org/manual/reference/explain`.

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

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

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

For more information, see *Indexing Strategies* (page 199).

Query Operations that Cannot Use Indexes Effectively

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

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

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

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

Cursors

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

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

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

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

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

  myCursor
  ```

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

[3] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *mongo-shell-executing-queries* for more information.

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

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

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

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

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

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

myCursor.forEach(printjson);
```

See *JavaScript cursor methods* and your `driver` documentation for more information on cursor methods.

Iterator Index

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

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

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

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

Consider the following example:

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

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

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

Cursor Behaviors

Consider the following behaviors related to cursors:

- By default, the server will automatically close the cursor after 10 minutes of inactivity or if client has exhausted the cursor. To override this behavior, you can specify the `noTimeout` wire protocol flag[4] in your query;

[4] http://docs.mongodb.org/meta-driver/latest/legacy/mongodb-wire-protocol

however, you should either close the cursor manually or exhaust the cursor. In the `mongo` shell, you can set the `noTimeout` flag:

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

See your `driver` documentation for information on setting the `noTimeout` flag. See *Cursor Flags* (page 35) for a complete list of available cursor flags.

- Because the cursor is not isolated during its lifetime, intervening write operations may result in a cursor that returns a single document [5] more than once. To handle this situation, see the information on *snapshot mode* (page 366).
- The MongoDB server returns the query results in batches:
  - For most queries, the *first* batch returns 101 documents or just enough documents to exceed 1 megabyte. Subsequent batch size is 4 megabytes. To override the default size of the batch, see `batchSize()` and `limit()`.
  - For queries that include a sort operation *without* an index, the server must load all the documents in memory to perform the sort and will return all documents in the first batch.
  - Batch size will not exceed the *maximum BSON document size*.
  - As you iterate through the cursor and reach the end of the returned batch, if there are more results, `cursor.next()` will perform a `getmore operation` to retrieve the next batch.

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

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

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

    myCursor.objsLeftInBatch();
    ```

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

  Consider the following example:

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

  The result from the command returns the following documentation:

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

Cursor Flags

The `mongo` shell provides the following cursor flags:

- `DBQuery.Option.tailable`
- `DBQuery.Option.slaveOk`
- `DBQuery.Option.oplogReplay`

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

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

Aggregation

Changed in version 2.2.

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

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

The aggregation framework provides users with a "pipeline" like framework: documents enter from a collection and then pass through a series of steps by a sequence of *pipeline operators* (page 164) that manipulate and transform the documents until they're output at the end. The aggregation framework is accessible via the aggregate command or the db.collection.aggregate() helper in the mongo shell.

For more information on the aggregation framework see *Aggregation* (page 151).

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

- count (count())
- distinct (db.collection.distinct())
- group (db.collection.group())
- mapReduce. (Also consider mapReduce() and *Map-Reduce* (page 174).)

Architecture

Read Operations from Sharded Clusters

Sharded clusters allow you to partition a data set among a cluster of mongod in a way that is nearly transparent to the application. See the *Sharding* (page 295) section of this manual for additional information about these deployments.

For a sharded cluster, you issue all operations to one of the mongos instances associated with the cluster. mongos instances route operations to the mongod in the cluster and behave like mongod instances to the application. Read operations to a sharded collection in a sharded cluster are largely the same as operations to a *replica set* or *standalone* instances. See the section on *Read Operations in Sharded Clusters* (page 299) for more information.

In sharded deployments, the mongos instance routes the queries from the clients to the mongod instances that hold the data, using the cluster metadata stored in the *config database* (page 298).

For sharded collections, if queries do not include the *shard key* (page 296), the mongos must direct the query to all shards in a collection. These *scatter gather* queries can be inefficient, particularly on larger clusters, and are unfeasible for routine operations.

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

- *An Introduction to Shard Keys* (page 296)
- *Shard Key Internals and Operations* (page 304)
- *Querying Sharded Clusters* (page 306)
- *Sharded Cluster Operations and mongos Instances* (page 299)

Read Operations from Replica Sets

Replica sets use *read preferences* to determine where and how to route read operations to members of the replica set. By default, MongoDB always reads data from a replica set's *primary*. You can modify that behavior by changing the *read preference mode* (page 244).

You can configure the *read preference mode* (page 244) 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 236) situations.

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

For more information on *read preferences* (page 243) or on the read preference modes, see *Read Preference* (page 243) and *Read Preference Modes* (page 244).

2.1.2 Write Operations

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

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

Write Operators

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

- *Create* (page 61)
- *Update* (page 77)
- *Delete* (page 83)

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

- `db.collection.insert()`
- `db.collection.update()`
- `db.collection.save()`
- `db.collection.findAndModify()`

- `db.collection.remove()`

For information on how to perform write operations from within an application, see the `http://docs.mongodb.org/manual/applications/drivers` documentation or the documentation for your client library.

Write Concern

Note: The *driver write concern* (page 419) change created a new connection class in all of the MongoDB drivers, called `MongoClient` with a different default write concern. See the *release notes* (page 419) for this change, and the release notes for the driver you're using for more information about your driver's release.

Operational Considerations and Write Concern

Clients issue write operations with some level of *write concern*, which describes the level of concern or guarantee the server will provide in its response to a write operation. Consider the following levels of conceptual write concern:

- *errors ignored*: Write operations are not acknowledged by MongoDB, and may not succeed in the case of connection errors that the client is not yet aware of, or if the `mongod` produces an exception (e.g. a duplicate key exception for *unique indexes* (page 190).) While this operation is efficient because it does not require the database to respond to every write operation, it also incurs a significant risk with regards to the persistence and durability of the data.

  Warning: Do not use this option in normal operation.

- *unacknowledged*: MongoDB does not acknowledge the receipt of write operation as with a write concern level of *ignore*; however, the driver will receive and handle network errors, as possible given system networking configuration.

  Before the releases outlined in *Default Write Concern Change* (page 419), this was the default write concern.

- receipt *acknowledged*: The `mongod` will confirm the receipt of the write operation, allowing the client to catch network, duplicate key, and other exceptions. After the releases outlined in *Default Write Concern Change* (page 419), this is the default write concern. [6]

- *journaled*: The `mongod` will confirm the write operation only after it has written the operation to the *journal*. This confirms that the write operation can survive a `mongod` shutdown and ensures that the write operation is durable.

  While receipt *acknowledged* without *journaled* provides the fundamental basis for write concern, there is a window between journal commits where the write operation is not fully durable. See `journalCommitInterval` for more information on this window. Require *journaled* as part of the write concern to provide this durability guarantee.

Replica sets present an additional layer of consideration for write concern. Basic write concern levels affect the write operation on only one `mongod` instance. The `w` argument to `getLastError` provides a *replica acknowledged* level of write concern. With *replica acknowledged* you can guarantee that the write operation has propagated to the members of a replica set. See the *Write Concern for Replica Sets* (page 241) document for more information.

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

[6] The default write concern is to call `getLastError` with no arguments. For replica sets, you can define the default write concern settings in the `getLastErrorDefaults` If `getLastErrorDefaults` does not define a default write concern setting, `getLastError` defaults to basic receipt acknowledgment.

Internal Operation of Write Concern

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

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

The definition of a "successful write" depends on the arguments specified to `getLastError`, or in replica sets, the configuration of `getLastErrorDefaults`. When deciding the level of write concern for your application, become familiar with the *Operational Considerations and Write Concern* (page 38).

The `getLastError` command has the following options to configure write concern requirements:

- `j` or "journal" option

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

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

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

- `w` option

  This option provides the ability to disable write concern entirely *as well as* specifies the write concern operations for *replica sets*. See *Operational Considerations and Write Concern* (page 38) for an introduction to the fundamental concepts of write concern. By default, the `w` option is set to `1`, which provides basic receipt acknowledgment on a single `mongod` instance or on the *primary* in a replica set.

  The `w` option takes the following values:

  - `-1`:

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

  - `0`:

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

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

  - `1`:

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

  - *A number greater than 1*:

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

  - `majority`:

    Confirms that write operations have propagated to the majority of configured replica set: nodes must acknowledge the write operation before it succeeds. This ensures that write operation will *never* be subject

to a rollback in the course of normal operation, and furthermore allows you to prevent hard coding assumptions about the size of your replica set into your application.

– *A tag set*:

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

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

For more information on write concern and replica sets, see *Write Concern for Replica Sets* (page 241) for more information..

In sharded clusters, `mongos` instances will pass write concern on to the shard `mongod` instances.

Bulk Inserts

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

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

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

Bulk insert can significantly increase performance by amortizing *write concern* (page 38) costs. In the drivers, you can configure write concern for batches rather than on a per-document level.

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

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

If the bulk insert process generates more than one error in a batch job, the client will only receive the most recent error. All bulk operations to a *sharded collection* run with `ContinueOnError`, which applications cannot disable. See *Strategies for Bulk Inserts in Sharded Clusters* (page 324) section for more information on consideration for bulk inserts in sharded clusters.

For more information see your `driver documentation` for details on performing bulk inserts in your application. Also consider the following resources: *Sharded Clusters* (page 43), *Strategies for Bulk Inserts in Sharded Clusters* (page 324), and *Importing and Exporting MongoDB Data* (page 117).

Indexing

After every insert, update, or delete operation, MongoDB must update *every* index associated with the collection in addition to the data itself. Therefore, every index on a collection adds some amount of overhead for the performance of write operations. [7]

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

For more information on indexes in MongoDB consider *Indexes* (page 185) and *Indexing Strategies* (page 199).

[7] The overhead for *sparse indexes* (page 190) inserts and updates to un-indexed fields is less than for non-sparse indexes. Also for non-sparse indexes, updates that don't change the record size have less indexing overhead.

Isolation

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

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

To isolate a sequence of write operations from other read and write operations, see `http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits`.

Updates

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

Documents in MongoDB can grow up to the full maximum `BSON document size`. However, when documents outgrow their allocated record size MongoDB must allocate a new record and move the document to the new record. Update operations that do not cause a document to grow, (i.e. *in-place* updates,) are significantly more efficient than those updates that cause document growth. Use *data models* (page 43) that minimize the need for document growth when possible.

For complete examples of update operations, see *Update* (page 77).

Padding Factor

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

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

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

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

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

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

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

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

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

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

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

Note: The following operations remove padding:

- `compact`,
- `repairDatabase`, and
- initial replica sync operations.

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

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

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

See also:

- *Can I manually pad documents to prevent moves during updates?* (page 367)
- Fast Updates with MongoDB with in-place Updates[8] (blog post)

Architecture

Replica Sets

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

Large volumes of write operations, particularly bulk operations, may create situations where the secondary members have difficulty applying the replicating operations from the primary at a sufficient rate: this can cause the secondary's state to fall behind that of the primary. Secondaries that are significantly behind the primary present problems for normal operation of the replica set, particularly *failover* (page 218) in the form of *rollbacks* (page 219) as well as general *read consistency* (page 219).

To help avoid this issue, you can customize the *write concern* (page 38) to return confirmation of the write operation to another member [9] 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 *Write Concern* (page 241), *Oplog* (page 220), *Oplog Internals* (page 250), and *Changing Oplog Size* (page 231).

[8] http://blog.mongodb.org/post/248614779/fast-updates-with-mongodb-update-in-place

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

Sharded Clusters

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

For more information, see *Sharded Cluster Administration* (page 298) and *Bulk Inserts* (page 40).

2.2 Document Orientation Concepts

2.2.1 Data Modeling Considerations for MongoDB Applications

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. This means that:

- documents in the same collection do not need to have the same set of fields or structure, and
- common fields in a collection's documents may hold different types of data.

Each document only needs to contain relevant fields to the entity or object that the document represents. In practice, *most* documents in a collection share a similar structure. Schema flexibility means that you can model your documents in MongoDB so that they can closely resemble and reflect application-level objects.

As in all data modeling, when developing data models (i.e. *schema designs,*) for MongoDB you must consider the inherent properties and requirements of the application objects and the relationships between application objects. MongoDB data models must also reflect:

- how data will grow and change over time, and
- the kinds of queries your application will perform.

These considerations and requirements force developers to make a number of multi-factored decisions when modeling data, including:

- normalization and de-normalization.

  These decisions reflect degree to which the data model should store related pieces of data in a single document **or** should the data model describe relationships using *references* (page 56) between documents.

- *indexing strategy* (page 199).
- representation of data in arrays in *BSON*.

Although a number of data models may be functionally equivalent for a given application; however, different data models may have significant impacts on MongoDB and applications performance.

This document provides a high level overview of these data modeling decisions and factors. In addition, consider, the *Data Modeling Patterns and Examples* (page 46) section which provides more concrete examples of all the discussed patterns.

Data Modeling Decisions

Data modeling decisions involve determining how to structure the documents to model the data effectively. The primary decision is whether to *embed* (page 44) or to *use references* (page 44).

Embedding

To de-normalize data, store two related pieces of data in a single *document*.

Operations within a document are less expensive for the server than operations that involve multiple documents.

In general, use embedded data models when:

- you have "contains" relationships between entities. See *Model Embedded One-to-One Relationships Between Documents* (page 84).
- you have one-to-many relationships where the "many" objects always appear with or are viewed in the context of their parent documents. See *Model Embedded One-to-Many Relationships Between Documents* (page 85).

Embedding provides the following benefits:

- generally better performance for read operations.
- the ability to request and retrieve related data in a single database operation.

Embedding related data in documents, can lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size`. For larger documents, consider using *GridFS* (page 58).

See also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 28) for more examples on accessing arrays.
- *Subdocuments* (page 27) for more examples on accessing subdocuments.

Referencing

To normalize data, store *references* (page 56) between two documents to indicate a relationship between the data represented in each document.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets. See *data-modeling-trees*.

Referencing provides more flexibility than embedding; however, to resolve the references, client-side applications must issue follow-up queries. In other words, using references requires more roundtrips to the server.

See *Model Referenced One-to-Many Relationships Between Documents* (page 87) for an example of referencing.

Atomicity

MongoDB only provides atomic operations on the level of a single document. [10] As a result needs for atomic operations influence decisions to use embedded or referenced relationships when modeling data for MongoDB.

Embed fields that need to be modified together atomically in the same document. See *Model Data for Atomic Operations* (page 88) for an example of atomic updates within a single document.

[10] Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple subdocuments within that single record are still atomic.

Operational Considerations

In addition to normalization and normalization concerns, a number of other operational factors help shape data modeling decisions in MongoDB. These factors include:

- data lifecycle management,
- number of collections and
- indexing requirements,
- sharding, and
- managing document growth.

These factors implications for database and application performance as well as future maintenance and development costs.

Data Lifecycle Management

Data modeling decisions should also take data lifecycle management into consideration.

The `Time to Live or TTL feature` 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 `http://docs.mongodb.org/manual/core/capped-collections`. Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and optimized to support operations that insert and read documents based on insertion order.

Large Number of Collections

In certain situations, you might choose to store information in several collections rather than in a single collection.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

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

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

Generally, having large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.

A single `<database>.ns` file stores all meta-data for each *database*. Each index and collection has its own entry in the namespace file, MongoDB places `limits on the size of namespace files`.

Because of `limits on namespaces`, you may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support, as in the following example:

```
db.system.namespaces.count()
```

The <database>.ns file defaults to 16 MB. To change the size of the <database>.ns file, pass a new size to *--nssize option <new size MB>* on server start.

The *--nssize* sets the size for *new* <database>.ns files. For existing databases, after starting up the server with *--nssize*, run the db.repairDatabase() command from the mongo shell.

Indexes

Create indexes to support common queries. Generally, indexes and index use in MongoDB correspond to indexes and index use in relational database: build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the _id field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive as each insert must add keys to each index.
- Collections with high proportion of read operations to write operations often benefit from additional indexes. Indexes do not affect un-indexed read operations.

See *Indexing Strategies* (page 199) for more information on determining indexes. Additionally, the MongoDB database profiler may help identify inefficient queries.

Sharding

Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of mongod instances or *shards*.

The shard key determines how MongoDB distributes data among shards in a sharded collection. Selecting the proper *shard key* (page 296) has significant implications for performance.

See *Sharded Cluster Overview* (page 295) for more information on sharding and the selection of the *shard key* (page 296).

Document Growth

Certain updates to documents can increase the document size, such as pushing elements to an array and adding new fields. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. This internal relocation can be both time and resource consuming.

Although MongoDB automatically provides padding to minimize the occurrence of relocations, you may still need to manually handle document growth. Refer to Pre-Aggregated Reports Use Case Study[11] for an example of the *Pre-allocation* approach to handle document growth.

Data Modeling Patterns and Examples

The following documents provide overviews of various data modeling patterns and common schema design considerations:

- *Model Embedded One-to-One Relationships Between Documents* (page 84)

[11] http://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports

- *Model Embedded One-to-Many Relationships Between Documents* (page 85)
- *Model Referenced One-to-Many Relationships Between Documents* (page 87)
- *Model Data for Atomic Operations* (page 88)
- *Model Tree Structures with Parent References* (page 89)
- *Model Tree Structures with Child References* (page 90)
- *Model Tree Structures with Materialized Paths* (page 91)
- *Model Tree Structures with Nested Sets* (page 92)

For more information and examples of real-world data modeling, consider the following external resources:

- Schema Design by Example[12]
- Walkthrough MongoDB Data Modeling[13]
- Document Design for MongoDB[14]
- Dynamic Schema Blog Post[15]
- MongoDB Data Modeling and Rails[16]
- Ruby Example of Materialized Paths[17]
- Sean Cribs Blog Post[18] which was the source for much of the *data-modeling-trees* content.

2.2.2 BSON Documents

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

- the *records* (page 49) stored in *collections*
- the *query selectors* (page 51) that determine which records to select for read, update, and delete operations
- the *update actions* (page 51) that specify the particular field updates to perform during an update operation
- the specification of *indexes* (page 52) for collection.
- arguments to several MongoDB methods and operators, including:
  - *sort order* (page 52) for the `sort()` method.
  - *index specification* (page 52) for the `hint()` method.
- the output of a number of MongoDB commands and operations, including:
  - the `output` of `collStats` command, and
  - the `output` of the `serverStatus` command.

[12] http://www.mongodb.com/presentations/mongodb-melbourne-2012/schema-design-example
[13] http://blog.fiesta.cc/post/11319522700/walkthrough-mongodb-data-modeling
[14] http://oreilly.com/catalog/0636920018391
[15] http://dmerr.tumblr.com/post/6633338010/schemaless
[16] http://docs.mongodb.org/ecosystem/tutorial/model-data-for-ruby-on-rails/
[17] http://github.com/banker/newsmonger/blob/master/app/models/comment.rb
[18] http://seancribbs.com/tech/2009/09/28/modeling-a-tree-in-a-document-database

Structure

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

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

Having support for the full range of BSON types, MongoDB documents may contain field and value pairs where the value can be another document, an array, an array of documents as well as the basic types such as `Double`, `String`, and `Date`. See also *BSON Type Considerations* (page 53).

Consider the following document that contains values of varying types:

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

The document contains the following fields:

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

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

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

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

Type Operators

To determine the type of fields, the `mongo` shell provides the following operators:

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

Example

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

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

  ```
  mydoc._id instanceof ObjectId
  ```

  The operation returns `true`.

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

  ```
  typeof mydoc._id
  ```

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

Dot Notation

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

To access an element of an array by the zero-based index position, you concatenate the array name with the dot (`.`) and zero-based index position:

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

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

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

See also:

- *Subdocuments* (page 27) for dot notation examples with subdocuments.
- *Arrays* (page 28) for dot notation examples with arrays.

Document Types in MongoDB

Record Documents

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

These documents have the following attributes:

- The maximum BSON document size is 16 megabytes.

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

- *Documents* (page 47) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` will add the `_id` field and generate the `ObjectId`.

The following document specifies a record in a collection:

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

The document contains the following fields:

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

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

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

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

- Use an `ObjectId`. See the *ObjectId* (page 54) documentation.

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

- Generate a sequence number for the documents in your collection in your application and use this value for the `_id` value. See the `http://docs.mongodb.org/manual/tutorial/create-an-auto-incrementing-field` tutorial for an implementation pattern.

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

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

  - the binary subtype value is in the range of 0-7 or 128-135, and
  - the length of the byte array is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, or 32.

- Use your driver's BSON UUID facility to generate UUIDs. Be aware that driver implementations may implement UUID serialization and deserialization logic differently, which may not be fully compatible with other drivers. See your driver documentation[19] for information concerning UUID interoperability.

Query Specification Documents

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

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

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

See also:

- *Query Document* (page 26) and *Read* (page 69) for more examples on selecting documents for reads.
- *Update* (page 77) for more examples on selecting documents for updates.
- *Delete* (page 83) for more examples on selecting documents for deletes.

Update Specification Documents

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

Consider the update document example:

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

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

- Modifies the field `name` whose value is another document. Specifically, the `$set` operator updates the `middle` field in the `name` subdocument. The document uses *dot notation* (page 49) to access a field in a subdocument.
- Adds an element to the field `awards` whose value is an array. Specifically, the `$push` operator adds another document as element to the field `awards`.

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

[19] http://api.mongodb.org/

```
                        by: 'IBM'
                    }
            }
    }
)
```

See also:

- *update operators* page for the available update operators and syntax.
- *update* (page 77) for more examples on update documents.

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

Index Specification Documents

Index specification documents describe the fields to index on during the `index creation`. See *indexes* (page 185) for an overview of indexes. [20]

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

```
{ field: value }
```

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

The following document specifies the *multi-key index* (page 189) on the `_id` field and the `last` field contained in the subdocument `name` field. The document uses *dot notation* (page 49) to access a field in a subdocument:

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

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

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

Sort Order Specification Documents

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

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

```
{ field: value }
```

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

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

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

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

[20] Indexes optimize a number of key *read* (page 25) and *write* (page 37) operations.

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

BSON Type Considerations

The following BSON types require special consideration:

ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first 4-bytes is a timestamp that reflects the ObjectId's creation. Refer to the *ObjectId* (page 54) 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. [21] In addition, MongoDB `$regex` queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 54) type. Timestamp values are a 64 bit value where:

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

Within a single `mongod` instance, timestamp values are always unique.

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

Note: The BSON Timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See *Date* (page 54) for more information.

If you create a BSON Timestamp using the empty constructor (e.g. `new Timestamp()`), MongoDB will only generate a timestamp *if* you use the constructor in the first field of the document. [22] Otherwise, MongoDB will generate an empty timestamp value (i.e. `Timestamp(0, 0)`.)

Changed in version 2.1: `mongo` shell displays the Timestamp value with the wrapper:

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

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

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

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

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

Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). The official BSON specification[23] refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed. [24] Negative values represent dates before 1970.

Consider the following examples of BSON Date:

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

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

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

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

- Return the `Date` value as string:

```
mydate1.toString()
```

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

```
mydate1.getMonth()
```

2.2.3 ObjectId

Overview

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

- a 4-byte timestamp,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

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

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

- you can access the timestamp of the ObjectId's creation, using the `getTimestamp()` method.
- sorting on an `_id` field that stores `ObjectId` values is roughly equivalent to sorting by creation time, although this relationship is not strict with `ObjectId` values generated on multiple systems within a single second.

Also consider the *BSON Documents* (page 47) section for related information on MongoDB's document orientation.

[23] http://bsonspec.org/#/specification

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

ObjectId()

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

- str

  The hexadecimal string value of the ObjectId() object.

- getTimestamp()

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

- toString()

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

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

- valueOf()

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

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

Examples

Consider the following uses ObjectId() class in the mongo shell:

- To generate a new ObjectId, use the ObjectId() constructor with no argument:

  ```
  x = ObjectId()
  ```

  In this example, the value of x would be:

  ```
  ObjectId("507f1f77bcf86cd799439011")
  ```

- To generate a new ObjectId using the ObjectId() constructor with a unique hexadecimal string:

  ```
  y = ObjectId("507f191e810c19729de860ea")
  ```

  In this example, the value of y would be:

  ```
  ObjectId("507f191e810c19729de860ea")
  ```

- To return the timestamp of an ObjectId() object, use the getTimestamp() method as follows:

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

  This operation will return the following Date object:

  ```
  ISODate("2012-10-17T20:46:22Z")
  ```

- Access the str attribute of an ObjectId() object, as follows:

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

  This operation will return the following hexadecimal string:

```
507f191e810c19729de860ea
```

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

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

  This operation will return the following output:

  ```
  ObjectId("507f191e810c19729de860ea")
  ```

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

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

  This operation returns the following output:

  ```
  507f191e810c19729de860ea
  ```

2.2.4 Database References

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

MongoDB applications use one of two methods for relating documents:

1. *Manual references* (page 56) where you save the `_id` field of one document in another document as a reference. Then your application can run a second query to return the embedded data. These references are simple and sufficient for most use cases.
2. *DBRefs* (page 57) are references from one document to another using the value of the first document's `_id` field collection, and optional database name. To resolve DBRefs, your application must perform additional queries to return the referenced documents. Many `drivers` have helper methods that form the query for the DBRef automatically. The drivers [25] do not *automatically* resolve DBRefs into documents.

   Use a DBRef when you need to embed documents from multiple collections in documents from one collection. DBRefs also provide a common format and type to represent these relationships among documents. The DBRef format provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason for using a DBRef, use manual references.

Manual References

Background

Manual references refers to the practice of including one *document's* `_id` field in another document. The application can then issue a second query to resolve the referenced fields as needed.

Process

Consider the following operation to insert two documents, using the `_id` field of the first document as a reference in the second document:

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

```
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 56). The references are simple to create and your application can resolve references as needed.

The only limitation of manual linking is that these references do not convey the database and collection name. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using *DBRefs* (page 57).

DBRefs

Background

DBRefs are a convention for representing a *document*, rather than a specific reference "type." They include the name of the collection, and in some cases the database, in addition to the value from the `_id` field.

Format

DBRefs have the following fields:

`$ref`
: The `$ref` field holds the name of the collection where the referenced document resides.

`$id`
: The `$id` field contains the value of the `_id` field in the referenced document.

`$db`
: *Optional.*

  Contains the name of the database where the referenced document resides.

  Only some drivers support `$db` references.

Example

DBRef document would resemble the following:

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

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

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

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

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

Support

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

C# The C# driver provides access to DBRef objects with the MongoDBRef Class[26] and supplies the FetchDBRef Method[27] for accessing these objects.

Java The DBRef[28] class provides supports for DBRefs from Java.

JavaScript The `mongo` shell's `JavaScript` interface provides a DBRef.

Perl The Perl driver contains no support for DBRefs. You can transverse references manually or use the MongoDBx::AutoDeref[29] CPAN module.

PHP The PHP driver does support DBRefs, including the optional `$db` reference, through The MongoDBRef class[30].

Python The Python driver provides the DBRef class[31], and the dereference method[32] for interacting with DBRefs.

Ruby The Ruby Driver supports DBRefs using the DBRef class[33] and the deference method[34].

Use

In most cases you should use the *manual reference* (page 56) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

2.2.5 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, [35] and stores each of those

[26] http://api.mongodb.org/csharp/current/html/46c356d3-ed06-a6f8-42fa-e0909ab64ce2.htm
[27] http://api.mongodb.org/csharp/current/html/1b0b8f48-ba98-1367-0a7d-6e01c8df436f.htm
[28] http://api.mongodb.org/java/current/com/mongodb/DBRef.html
[29] http://search.cpan.org/dist/MongoDBx-AutoDeref/
[30] http://www.php.net/manual/en/class.mongodbref.php/
[31] http://api.mongodb.org/python/current/api/bson/dbref.html
[32] http://api.mongodb.org//python/current/api/pymongo/database.html#pymongo.database.Database.dereference
[33] http://api.mongodb.org//ruby/current/BSON/DBRef.html
[34] http://api.mongodb.org//ruby/current/Mongo/DB.html#dereference
[35] 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.

chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

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

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory. For more information on the indications of GridFS, see *When should I use GridFS?* (page 362).

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 `http://docs.mongodb.org/manual/reference/mongofiles`.

GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 59).
- `files` stores the file's metadata. For details, see *The files Collection* (page 60).

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.

The `chunks` Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <string>,
  "files_id" : <string>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

`chunks.`**`_id`**
: The unique *ObjectId* of the chunk.

`chunks.`**`files_id`**
: The `_id` of the "parent" document, as specified in the `files` collection.

`chunks.`**`n`**
: The sequence number of the chunk. GridFS numbers all chunks, starting with 0.

`chunks.data`
: The chunk's payload as a *BSON* binary type.

The `chunks` collection uses a *compound index* on `files_id` and `n`, as described in *GridFS Index* (page 60).

The `files` Collection

Each document in the `files` collection represents a file in the *GridFS* store. Consider the following prototype of a document in the `files` collection:

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>
  "uploadDate" : <timestamp>
  "md5" : <hash>

  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

Documents in the `files` collection contain some or all of the following fields. Applications may create additional arbitrary fields:

`files._id`
: The unique ID for this document. The `_id` is of the data type you chose for the original document. The default type for MongoDB documents is *BSON ObjectId*.

`files.length`
: The size of the document in bytes.

`files.chunkSize`
: The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 kilobytes.

`files.uploadDate`
: The date the document was first stored by GridFS. This value has the `Date` type.

`files.md5`
: An MD5 hash returned from the filemd5 API. This value has the `String` type.

`files.filename`
: Optional. A human-readable name for the document.

`files.contentType`
: Optional. A valid MIME type for the document.

`files.aliases`
: Optional. An array of alias strings.

`files.metadata`
: Optional. Any additional information you want to store.

GridFS Index

GridFS uses a *unique*, *compound* index on the `chunks` collection for `files_id` and `n`. The index allows efficient retrieval of chunks using the `files_id` and `n` values, as shown in the following example:

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

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

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

Example Interface

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

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

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

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

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

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

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

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

2.3 CRUD Operations for MongoDB

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

2.3.1 Create

Of the four basic database operations (i.e. CRUD), *create* operations are those that add new records or *documents* to a *collection* in MongoDB. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 37); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 25) page.

Overview

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

- *insert* (page 62)
- *updates with the upsert option* (page 67)

All insert operations in MongoDB exhibit the following properties:

- If you attempt to insert a document without the *_id* field, the client library *or* the `mongod` instance will add an `_id` field and populate the field with a unique *ObjectId*.

- For operations with *write concern* (page 38), if you specify an `_id` field, the `_id` field must be unique within the collection; otherwise the `mongod` will return a duplicate key exception.
- The maximum BSON document size is 16 megabytes.

  The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` and the documentation for your `driver` for more information about GridFS.
- *Documents* (page 47) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.

Note: As of these *driver versions* (page 420), all write operations will issue a `getLastError` command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 38) in the *Write Operations* (page 37) document for more information.

`insert()`

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

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

Corresponding Operation in SQL

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

Insert the First Document in a Collection

If the collection does not exist [36], then the `insert()` method creates the collection during the first insert. Specifically in the example, if the collection `bios` does not exist , then the insert operation will create this collection:

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

[36] You can also view a list of the existing collections in the database using the `show collections` operation in the `mongo` shell.

```
                },
                {
                  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'
                }
              ]
   }
)
```

You can confirm the insert by *querying* (page 69) the `bios` collection:

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

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

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

Insert a Document without Specifying an _id Field

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

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

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

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

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

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

```
            ]
}
```

Bulk Insert Multiple Documents

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

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

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

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

Insert a Document with `save()`

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

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

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

```
                    year: 2003,
                    by: 'NLUUG'
                  }
                ]
    }
)
```

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

The `update()` operation in MongoDB accepts an "`upsert`" flag that modifies the behavior of `update()` from *updating existing documents* (page 77), to inserting data.

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

- If the query matches an existing document(s), the operation is an *update* (page 77).
- If the query matches no document in the collection, the operation is an *insert* (page 61).

An upsert operation has the following syntax [37]:

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

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

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

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

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

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

```
                award: 'Japan Prize',
                year: 2011,
                by: 'The Japan Prize Foundation'
              }
            ]
    },
    { upsert: true }
)
```

Insert a Document that Contains Update Operator Expressions

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

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

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

Update operations with `save()`

The `save()` method is identical to an *update operation with the upsert flag* (page 67)

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

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

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

2.3.2 Read

Of the four basic database operations (i.e. CRUD), read operations are those that retrieve records or *documents* from a *collection* in MongoDB. For general information about read operations and the factors that affect their performance, see *Read Operations* (page 25); for documentation of the other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 25) page.

Overview

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

- *find* (page 69)
- *findOne* (page 76)

`find()`

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

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

Corresponding Operation in SQL

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

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

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

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

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

Return All Documents in a Collection

If there is no <query> argument, the :method:‘ ~db.collection.find()‘ method selects all documents from a collection.

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

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

Return Documents that Match Query Conditions

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

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

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

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

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

On Arrays

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

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

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

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

On Subdocuments

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

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

```
    }
)
```

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

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

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

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

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

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

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

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

Logical Operators

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

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

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

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

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

With a Projection

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

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

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

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

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

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

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

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

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

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

See also:

- *dot notation* for information on "reaching into" embedded sub-documents.
- *Arrays* (page 28) for more examples on accessing arrays.
- *Subdocuments* (page 27) for more examples on accessing subdocuments.
- `$elemMatch` query operator for more information on matching array elements.
- `$elemMatch` projection operator for additional information on restricting array elements to return.

Iterate the Returned Cursor

The `find()` method returns a *cursor* to the results; however, in the `mongo` shell, if the returned cursor is not assigned to a variable, then the cursor is automatically iterated up to 20 times [38] to print up to the first 20 documents that match the query, as in the following example:

```
db.bios.find( { _id: 1 } );
```

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

```
var myCursor = db.bios.find( { _id: 1 } );

myCursor
```

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

```
var myCursor = db.bios.find( { _id: 1 } );

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

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

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

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

[38] You can use the `DBQuery.shellBatchSize` to change the number of iteration from the default value `20`. See *Cursor Flags* (page 35) and *Cursor Behaviors* (page 34) for more information.

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

```
var myCursor = db.bios.find( { _id: 1 } );

myCursor.forEach(printjson);
```

For more information on cursor handling, see:

- `cursor.hasNext()`
- `cursor.next()`
- `cursor.forEach()`
- *cursors* (page 33)
- *JavaScript cursor methods*

Modify the Cursor Behavior

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

Order Documents in the Result Set The `sort()` method orders the documents in the result set.

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

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

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

Limit the Number of Documents to Return The `limit()` method limits the number of documents in the result set.

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

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

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

Set the Starting Point of the Result Set The `skip()` method controls the starting point of the results set.

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

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

Combine Cursor Methods You can chain these cursor methods, as in the following examples [39]:

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

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

See the *JavaScript cursor methods* reference and your `driver` documentation for additional references. See *Cursors* (page 33) for more information regarding cursors.

`findOne()`

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

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

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

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

With Empty Query Specification

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

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

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

With a Query Specification

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

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

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

With a Projection

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

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

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

Return All but the Excluded Fields The following operation returns a document in the `bios` collection where the `contribs` field contains the element `OOP` and returns all fields *except* the `_id` field, the `first` field in the `name` subdocument, and the `birth` field from the matching documents:

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

Access the `findOne` Result

Although similar to the `find()` method, because the `findOne()` method returns a document rather than a cursor, you cannot apply the cursor methods such as `limit()`, `sort()`, and `skip()` to the result of the `findOne()` method. However, you can access the document directly, as in the example:

```
var myDocument = db.bios.findOne();

if (myDocument) {
   var myName = myDocument.name;

   print (tojson(myName));
}
```

2.3.3 Update

Of the four basic database operations (i.e. CRUD), *update* operations are those that modify existing records or *documents* in a MongoDB *collection*. For general information about write operations and the factors that affect their performance, see *Write Operations* (page 37); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 25) page.

Overview

Update operation modifies an existing *document* or documents in a *collection*. MongoDB provides the following methods to perform update operations:

- *update* (page 78)
- *save* (page 81)

Note: Consider the following behaviors of MongoDB's update operations.

- When performing update operations that increase the document size beyond the allocated space for that document, the update operation relocates the document on disk and may reorder the document fields depending on the type of update.
- As of these *driver versions* (page 420), all write operations will issue a `getLastError` command to confirm the result of the write operation:

  ```
  { getLastError: 1 }
  ```

  Refer to the documentation on *write concern* (page 38) in the *Write Operations* (page 37) document for more information.

Update

The `update()` method is the primary method used to modify documents in a MongoDB collection. By default, the `update()` method updates a **single** document, but by using the `multi` option, `update()` can update all documents that match the query criteria in the collection. The `update()` method can either replace the existing document with the new document or update specific fields in the existing document.

The `update()` has the following syntax [40]:

```
db.collection.update( <query>, <update>, <options> )
```

Corresponding operation in SQL

The `update()` method corresponds to the `UPDATE` operation in SQL, and:

- the `<query>` argument corresponds to the `WHERE` statement, and
- the `<update>` corresponds to the `SET ...` statement.

The default behavior of the `update()` method updates a **single** document and would correspond to the SQL `UPDATE` statement with the `LIMIT 1`. With the `multi` option, `update()` method would correspond to the SQL `UPDATE` statement without the `LIMIT` clause.

Modify with Update Operators

If the `<update>` argument contains only *update operator* (page 82) expressions such as the `$set` operator expression, the `update()` method updates the corresponding fields in the document. To update fields in subdocuments, MongoDB uses *dot notation*.

Update a Field in a Document Use `$set` to update a value of a field.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and sets the value of the field `middle`, in the subdocument `name`, to `Warner`:

```
db.bios.update(
   { _id: 1 },
   {
     $set: { 'name.middle': 'Warner' },
   }
)
```

Add a New Field to a Document If the `<update>` argument contains fields not currently in the document, the `update()` method adds the new fields to the document.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and adds to that document a new `mbranch` field and a new `aka` field in the subdocument `name`:

```
db.bios.update(
   { _id: 3 },
   { $set: {
             mbranch: 'Navy',
             'name.aka': 'Amazing Grace'
           }
```

[40] This examples uses the interface added in MongoDB 2.2 to specify the `multi` and the `upsert` options in a document form.
Prior to version 2.2, in the `mongo` shell, you would specify the `upsert` and the `multi` options in the `update()` method as positional boolean options. See `update()` for details.

```
    }
)
```

Remove a Field from a Document If the <update> argument contains `$unset` operator, the `update()` method removes the field from the document.

The following operation queries the `bios` collection for the first document that has an `_id` field equal to `3` and removes the `birth` field from the document:

```
db.bios.update(
   { _id: 3 },
   { $unset: { birth: 1 } }
)
```

Update Arrays

Update an Element by Specifying Its Position If the update operation requires an update of an element in an array field, the `update()` method can perform the update using the position of the element and *dot notation*. Arrays in MongoDB are zero-based.

The following operation queries the `bios` collection for the first document with `_id` field equal to `1` and updates the second element in the `contribs` array:

```
db.bios.update(
   { _id: 1 },
   { $set: { 'contribs.1': 'ALGOL 58' } }
)
```

Update an Element without Specifying Its Position The `update()` method can perform the update using the `$` positional operator if the position is not known. The array field must appear in the `query` argument in order to determine which array element to update.

The following operation queries the `bios` collection for the first document where the `_id` field equals `3` and the `contribs` array contains an element equal to `compiler`. If found, the `update()` method updates the first matching element in the array to `A compiler` in the document:

```
db.bios.update(
       { _id: 3, 'contribs': 'compiler' },
       { $set: { 'contribs.$': 'A compiler' } }
    )
```

Update a Document Element without Specifying Its Position The `update()` method can perform the update of an array that contains subdocuments by using the positional operator (i.e. `$`) and the *dot notation*.

The following operation queries the `bios` collection for the first document where the `_id` field equals `6` and the `awards` array contains a subdocument element with the `by` field equal to `ACM`. If found, the `update()` method updates the `by` field in the first matching subdocument:

```
db.bios.update(
   { _id: 6, 'awards.by': 'ACM'  } ,
   { $set: { 'awards.$.by': 'Association for Computing Machinery' } }
)
```

Add an Element to an Array The following operation queries the `bios` collection for the first document that has an `_id` field equal to `1` and adds a new element to the `awards` field:

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

Update Multiple Documents If the `<options>` argument contains the `multi` option set to `true` or `1`, the `update()` method updates all documents that match the query.

The following operation queries the `bios` collection for all documents where the `awards` field contains a subdocument element with the `award` field equal to `Turing` and sets the `turing` field to `true` in the matching documents [41]:

```
db.bios.update(
   { 'awards.award': 'Turing' },
   { $set: { turing: true } },
   { multi: true }
)
```

Replace Existing Document with New Document

If the `<update>` argument contains only field and value pairs, the `update()` method *replaces* the existing document with the document in the `<update>` argument, except for the `_id` field.

The following operation queries the `bios` collection for the first document that has a `name` field equal to `{ first: 'John', last: 'McCarthy' }` and replaces all but the `_id` field in the document with the fields in the `<update>` argument:

```
db.bios.update(
   { name: { first: 'John', last: 'McCarthy' } },
   { name: { first: 'Ken', last: 'Iverson' },
     born: new Date('Dec 17, 1941'),
     died: new Date('Oct 19, 2004'),
     contribs: [ 'APL', 'J' ],
     awards: [
               { award: 'Turing Award',
                 year: 1979,
                 by: 'ACM' },
               { award: 'Harry H. Goode Memorial Award',
                 year: 1975,
                 by: 'IEEE Computer Society' },
               { award: 'IBM Fellow',
                 year: 1970,
                 by: 'IBM' }
             ]
   }
)
```

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

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

If you set the `upsert` option in the `<options>` argument to `true` or `1` and no existing document match the `<query>` argument, the `update()` method can insert a new document into the collection. [42]

The following operation queries the `bios` collection for a document with the `_id` field equal to `11` and the `name` field equal to `{ first: 'James', last: 'Gosling'}`. If the query selects a document, the operation performs an update operation. If a document is not found, `update()` inserts a new document containing the fields and values from `<query>` argument with the operations from the `<update>` argument applied. [43]

```
db.bios.update(
   { _id:11, name: { first: 'James', last: 'Gosling' } },
   {
     $set: {
              born: new Date('May 19, 1955'),
              contribs: [ 'Java' ],
              awards: [
                        {
                          award: 'The Economist Innovation Award',
                          year: 2002,
                          by: 'The Economist'
                        },
                        {
                          award: 'Officer of the Order of Canada',
                          year: 2007,
                          by: 'Canada'
                        }
                      ]
            }
   },
   { upsert: true }
)
```

See also *Update Operations with the Upsert Flag* (page 67) in the *Create* (page 61) document.

Save

The `save()` method performs a special type of `update()`, depending on the `_id` field of the specified document.

The `save()` method has the following syntax:

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

Behavior

If you specify a document with an `_id` field, `save()` performs an `update()` with the `upsert` option set: if an existing document in the collection has the same `_id`, `save()` updates that document, and inserts the document otherwise. If you do not specify a document with an `_id` field to `save()`, performs an `insert()` operation.

That is, `save()` method is equivalent to the `update()` method with the `upsert` option and a `<query>` argument with an `_id` field.

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

[43] If the `<update>` argument includes only field and value pairs, the new document contains the fields and values specified in the `<update>` argument. If the `<update>` argument includes only *update operators* (page 82), the new document contains the fields and values from `<query>` argument with the operations from the `<update>` argument applied.

Example

Consider the following psudocode explanation of `save()` as an illustration of its behavior:

```
function save( doc ) {
  if( doc["_id"] ) {
       update( {_id: doc["_id"] }, doc, { upsert: true } );
    }
  else {
       insert(doc);
  }
}
```

Save Performs an Update

If the `<document>` argument contains the `_id` field that exists in the collection, the `save()` method performs an update that replaces the existing document with the `<document>` argument.

The following operation queries the `bios` collection for a document where the `_id` equals `ObjectId("507c4e138fada716c89d0014")` and replaces the document with the `<document>` argument:

```
db.bios.save(
   {
     _id: ObjectId("507c4e138fada716c89d0014"),
     name: { first: 'Martin', last: 'Odersky' },
     contribs: [ 'Scala' ]
   }
)
```

See also:

Insert a Document with save() (page 66) and *Update operations with save()* (page 69) in the *Create* (page 61) section.

Update Operators

Fields

- `$inc`
- `$rename`
- `$set`
- `$unset`

Array

- `$`
- `$addToSet`
- `$pop`
- `$pullAll`
- `$pull`

- $pushAll
- $push

Bitwise

- $bit

Isolation

- $isolated

2.3.4 Delete

Of the four basic database operations (i.e. CRUD), *delete* operations are those that remove documents from a *collection* in MongoDB.

For general information about write operations and the factors that affect their performance, see *Write Operations* (page 37); for documentation of other CRUD operations, see the *Core MongoDB Operations (CRUD)* (page 25) page.

Overview

The *remove()* (page 83) method in the mongo shell provides this operation, as do corresponding methods in the drivers.

Note: As of these *driver versions* (page 420), all write operations will issue a getLastError command to confirm the result of the write operation:

```
{ getLastError: 1 }
```

Refer to the documentation on *write concern* (page 38) in the *Write Operations* (page 37) document for more information.

Use the remove() method to delete documents from a collection. The remove() method has the following syntax:

```
db.collection.remove( <query>, <justOne> )
```

Corresponding operation in SQL

The remove() method is analogous to the DELETE statement, and:

- the <query> argument corresponds to the WHERE statement, and
- the <justOne> argument takes a Boolean and has the same effect as LIMIT 1.

---

remove() deletes documents from the collection. If you do not specify a query, remove() removes all documents from a collection, but does not remove the indexes. [44]

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

---

[44] 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 All Documents that Match a Condition

If there is a <query> argument, the remove() method deletes from the collection all documents that match the argument.

The following operation deletes all documents from the bios collection where the subdocument name contains a field first whose value starts with G:

```
db.bios.remove( { 'name.first' : /^G/ } )
```

Remove a Single Document that Matches a Condition

If there is a <query> argument and you specify the <justOne> argument as true or 1, remove() only deletes a single document from the collection that matches the query.

The following operation deletes a single document from the bios collection where the turing field equals true:

```
db.bios.remove( { turing: true }, 1 )
```

Remove All Documents from a Collection

If there is no <query> argument, the remove() method deletes all documents from a collection. The following operation deletes all documents from the bios collection:

```
db.bios.remove()
```

Note: This operation is not equivalent to the drop() method.

Capped Collection

You cannot use the remove() method with a *capped collection*.

Isolation

If the <query> argument to the remove() method matches multiple documents in the collection, the delete operation may interleave with other write operations to that collection. For an unsharded collection, you have the option to override this behavior with the $isolated isolation operator, effectively isolating the delete operation from other write operations. To isolate the operation, include $isolated: 1 in the <query> parameter as in the following example:

```
db.bios.remove( { turing: true, $isolated: 1 } )
```

2.4 Data Modeling Patterns

2.4.1 Model Embedded One-to-One Relationships Between 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 Considerations for*

MongoDB Applications (page 43) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 44) 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` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA"
   zip: 12345
}
```

If the `address` data is frequently retrieved with the `name` information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the `address` data in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA"
              zip: 12345
            }
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

2.4.2 Model Embedded One-to-Many Relationships Between 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 Considerations for MongoDB Applications* (page 43) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 44) 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` contains a reference to the `parent`.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: 12345
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: 12345
}
```

If your application frequently retrieves the `address` data with the `name` information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the `address` data entities in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: 12345
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: 12345
                }
              ]
 }
```

With the embedded data model, your application can retrieve the complete patron information with one query.

2.4.3 Model Referenced One-to-Many Relationships Between 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 Considerations for MongoDB Applications* (page 43) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 44) between documents to describe relationships between connected data.

Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}

{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

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

```
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher_id: "oreilly"
}
```

2.4.4 Model Data for Atomic Operations

Pattern

Consider the following example that keeps a library book and its checkout information. The example illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.

Consider the following `book` document that stores the number of available copies for checkout and the current checkout information:

```
book = {
          _id: 123456789,
          title: "MongoDB: The Definitive Guide",
          author: [ "Kristina Chodorow", "Mike Dirolf" ],
          published_date: ISODate("2010-09-24"),
          pages: 216,
          language: "English",
          publisher_id: "oreilly",
          available: 3,
          checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
        }
```

You can use the `db.collection.findAndModify()` method to atomically determine if a book is available for checkout and update with the new checkout information. Embedding the `available` field and the `checkout` field within the same document ensures that the updates to these fields are in sync:

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

2.4.5 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 Considerations for MongoDB Applications* (page 43) 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 44) 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 example that models a tree of categories using *Parent References*:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "Postgres", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

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

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

  ```
  db.categories.ensureIndex( { parent: 1 } )
  ```

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

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

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

2.4.6 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 Considerations for MongoDB Applications* (page 43) 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 44) 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 example that models a tree of categories using *Child References*:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "Postgres", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "Postgres" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

- The query to retrieve the immediate children of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "Databases" } ).children
  ```

- You can create an index on the field `children` to enable fast search by the child nodes:

  ```
  db.categories.ensureIndex( { children: 1 } )
  ```

- You can query for a node in the `children` field to find its parent node as well as its siblings:

  ```
  db.categories.find( { children: "MongoDB" } )
  ```

The *Child References* pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.

2.4.7 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 Considerations for MongoDB Applications* (page 43) 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 44) 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 example that models a tree of categories using *Array of Ancestors*:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "
db.categories.insert( { _id: "Postgres", ancestors: [ "Books", "Programming", "Databases" ], parent:
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

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

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

  ```
  db.categories.ensureIndex( { ancestors: 1 } )
  ```

- You can query by the `ancestors` to find all its descendants:

  ```
  db.categories.find( { ancestors: "Programming" } )
  ```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the ancestors field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* pattern but is more straightforward to use.

2.4.8 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 Considerations for MongoDB Applications* (page 43) 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 example that models a tree of categories using *Materialized Paths* ; the path string uses the comma `,` as a delimiter:

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "Postgres", path: ",Books,Programming,Databases," } )
```

- You can query to retrieve the whole tree, sorting by the `path`:

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

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

  ```
  db.categories.find( { path: /,Programming,/ } )
  ```

- You can also retrieve the descendants of `Books` where the `Books` is also at the topmost level of the hierarchy:

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

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

  ```
  db.categories.ensureIndex( { path: 1 } )
  ```

  This index may improve performance, depending on the query:

  - For queries of the `Books` sub-tree (e.g. `http://docs.mongodb.org/manual/^,Books,/`) an index on the `path` field improves the query performance significantly.
  - For queries of the `Programming` sub-tree (e.g. `http://docs.mongodb.org/manual/,Programming,/`), or similar queries of sub-tress, where the node might be in the middle of the indexed string, the query must inspect the entire index.

    For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

2.4.9 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 Considerations for MongoDB Applications* (page 43) 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 example that models a tree of categories using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "Postgres", parent: "Databases", left: 8, right: 9 } )
```

You can query to retrieve the descendants of a node:

```
var databaseCategory = db.v.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } }
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

2.4.10 Model Data to Support Keyword Search

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 189), this pattern can support application's keyword search operations.

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 94) section for more information.

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 189) on the array and create queries that select values from the array.

Example

Suppose you have a collection of library volumes that you want to make searchable by topics. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the `Moby-Dick` volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
```

```
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

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

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and *multi-key indexes* (page 189); 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.

CHAPTER 3

Administration

The documentation in this section outlines core administrative tasks and practices that operators of MongoDB will want to consider. In addition to the core topics that follow, also consider the relevant documentation in other sections including: *Sharding* (page 295), *Replication* (page 217), and *Indexes* (page 185).

3.1 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/mongodb.conf`. Confirm this by checking the content of the `/etc/init.d/mongod` or `/etc/rc.d/mongod` script to insure that the *control scripts* start the `mongod` with the appropriate configuration file (see below.)

To start MongoDB instance using this configuration issue a command in the following form:

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

Modify the values in the `/etc/mongodb.conf` file on your system to control the configuration of your database instance.

3.1.1 Starting, Stopping, and Running the Database

Consider the following basic configuration:

```
fork = true
bind_ip = 127.0.0.1
port = 27017
quiet = true
dbpath = /srv/mongodb
logpath = /var/log/mongodb/mongod.log
logappend = true
journal = true
```

For most standalone servers, this is a sufficient base configuration. It makes several assumptions, but consider the following explanation:

- `fork` is `true`, which enables a *daemon* mode for `mongod`, which detaches (i.e. "forks") the MongoDB from the current session and allows you to run the database as a conventional server.
- `bind_ip` is `127.0.0.1`, which forces the server to only listen for requests on the localhost IP. Only bind to secure interfaces that the application-level systems can access with access control provided by system network filtering (i.e. "*firewall*").
- `port` is `27017`, which is the default MongoDB port for database instances. MongoDB can bind to any port. You can also filter access based on port using network filtering tools.

  Note: UNIX-like systems require superuser privileges to attach processes to ports lower than 1024.

- `quiet` is `true`. This disables all but the most critical entries in output/log file. In normal operation this is the preferable operation to avoid log noise. In diagnostic or testing situations, set this value to `false`. Use `setParameter` to modify this setting during run time.
- `dbpath` is `/srv/mongodb`, which specifies where MongoDB will store its data files. `/srv/mongodb` and `/var/lib/mongodb` are popular locations. The user account that `mongod` runs under will need read and write access to this directory.
- `logpath` is `/var/log/mongodb/mongod.log` which is where `mongod` will write its output. If you do not set this value, `mongod` writes all output to standard output (e.g. `stdout`.)
- `logappend` is `true`, which ensures that `mongod` does not overwrite an existing log file following the server start operation.
- `journal` is `true`, which enables *journaling*. Journaling ensures single instance write-durability. 64-bit builds of `mongod` enable journaling by default. Thus, this setting may be redundant.

Given the default configuration, some of these values may be redundant. However, in many situations explicitly stating the configuration increases overall system intelligibility.

3.1.2 Security Considerations

The following collection of configuration options are useful for limiting access to a `mongod` instance. Consider the following:

```
bind_ip = 127.0.0.1,10.8.0.10,192.168.4.24
nounixsocket = true
auth = true
```

Consider the following explanation for these configuration decisions:

- "`bind_ip`" has three values: `127.0.0.1`, the localhost interface; `10.8.0.10`, a private IP address typically used for local networks and VPN interfaces; and `192.168.4.24`, a private network interface typically used for local networks.

  Because production MongoDB instances need to be accessible from multiple database servers, it is important to bind MongoDB to multiple interfaces that are accessible from your application servers. At the same time it's important to limit these interfaces to interfaces controlled and protected at the network layer.
- "`nounixsocket`" to `true` disables the UNIX Socket, which is otherwise enabled by default. This limits access on the local system. This is desirable when running MongoDB on systems with shared access, but in most situations has minimal impact.
- "`auth`" is `true` enables the authentication system within MongoDB. If enabled you will need to log in by connecting over the `localhost` interface for the first time to create user credentials.

See also:

Security Practices and Management (page 133)

3.1.3 Replication and Sharding Configuration

Replication Configuration

Replica set configuration is straightforward, and only requires that the `replSet` have a value that is consistent among all members of the set. Consider the following:

```
replSet = set0
```

Use descriptive names for sets. Once configured use the `mongo` shell to add hosts to the replica set.

See also:

Replica set reconfiguration.

To enable authentication for the *replica set*, add the following option:

```
keyFile = /srv/mongodb/keyfile
```

New in version 1.8: for replica sets, and 1.9.1 for sharded replica sets.

Setting `keyFile` enables authentication and specifies a key file for the replica set member use to when authenticating to each other. The content of the key file is arbitrary, but must be the same on all members of the *replica set* and `mongos` instances that connect to the set. The keyfile must be less than one kilobyte in size and may only contain characters in the base64 set and the file must not have group or "world" permissions on UNIX systems.

See also:

The "*Replica set Reconfiguration*" section for information regarding the process for changing replica set during operation.

Additionally, consider the "*Replica Set Security* (page 232)" section for information on configuring authentication with replica sets.

Finally, see the "*Replication* (page 217)" index and the "*Replica Set Fundamental Concepts* (page 217)" document for more information on replication in MongoDB and replica set configuration in general.

Sharding Configuration

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

Note: *Config servers* are not *replica sets*.

To set up one or three "config server" instances as *normal* (page 95) `mongod` instances, and then add the following configuration option:

```
configsvr = true

bind_ip = 10.8.0.12
port = 27001
```

This creates a config server running on the private IP address `10.8.0.12` on port `27001`. Make sure that there are no port conflicts, and that your config server is accessible from all of your "`mongos`" and "`mongod`" instances.

To set up shards, configure two or more `mongod` instance using your *base configuration* (page 95), adding the `shardsvr` setting:

```
shardsvr = true
```

Finally, to establish the cluster, configure at least one `mongos` process with the following settings:

```
configdb = 10.8.0.12:27001
chunkSize = 64
```

You can specify multiple `configdb` instances by specifying hostnames and ports in the form of a comma separated list. In general, avoid modifying the `chunkSize` from the default value of 64, [1] and *should* ensure this setting is consistent among all `mongos` instances.

See also:

The "*Sharding* (page 295)" section of the manual for more information on sharding and cluster configuration.

3.1.4 Running Multiple Database Instances on the Same System

In many cases running multiple instances of `mongod` on a single system is not recommended. On some types of deployments [2] and for testing purposes you may need to run more than one `mongod` on a single system.

In these cases, use a *base configuration* (page 95) for each instance, but consider the following configuration values:

```
dbpath = /srv/mongodb/db0/
pidfilepath = /srv/mongodb/db0.pid
```

The `dbpath` value controls the location of the `mongod` instance's data directory. Ensure that each database has a distinct and well labeled data directory. The `pidfilepath` controls where `mongod` process places 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.

3.1.5 Diagnostic Configurations

The following configuration options control various `mongod` behaviors for diagnostic purposes. The following settings have default values that tuned for general production purposes:

```
slowms = 50
profile = 3
verbose = true
diaglog = 3
objcheck = true
cpu = true
```

Use the *base configuration* (page 95) and add these options if you are experiencing some unknown issue or performance problem as needed:

- `slowms` configures the threshold for the *database profiler* to consider a query "slow." The default value is 100 milliseconds. Set a lower value if the database profiler does not return useful results. See `http://docs.mongodb.org/manual/applications/optimization` for more information on optimizing operations in MongoDB.

[1] *Chunk* size is 64 megabytes by default, which provides the ideal balance between the most even distribution of data, for which smaller chunk sizes are best, and minimizing chunk migration, for which larger chunk sizes are optimal.

[2] Single-tenant systems with *SSD* or other high performance disks may provide acceptable performance levels for multiple `mongod` instances. Additionally, you may find that multiple databases with small working sets may function acceptably on a single system.

- `profile` sets the *database profiler* level. The profiler is not active by default because of the possible impact on the profiler itself on performance. Unless this setting has a value, queries are not profiled.
- `verbose` enables a verbose logging mode that modifies `mongod` output and increases logging to include a greater number of events. Only use this option if you are experiencing an issue that is not reflected in the normal logging level. If you require additional verbosity, consider the following options:

  ```
  v = true
  vv = true
  vvv = true
  vvvv = true
  vvvvv = true
  ```

  Each additional level `v` adds additional verbosity to the logging. The `verbose` option is equal to `v = true`.
- `diaglog` enables *diagnostic logging*. Level `3` logs all read and write options.
- `objcheck` forces `mongod` to validate all requests from clients upon receipt. Use this option to ensure that invalid requests are not causing errors, particularly when running a database with untrusted clients. This option may affect database performance.
- `cpu` forces `mongod` to report the percentage of the last interval spent in *write-lock*. The interval is typically 4 seconds, and each output line in the log includes both the actual interval since the last report and the percentage of time spent in write lock.

3.2 Operational Segregation in MongoDB Operations and Deployments

3.2.1 Operational Overview

MongoDB includes a cluster 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 `mongod` instances. MongoDB supports segmentation of operations across different dimensions, which may include multiple data centers and geographical regions in multi-data center deployments or 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 243), which controls how drivers help applications target read operations to members of a replica set.
- *Write Concerns* (page 241), which controls how MongoDB ensures that write operations propagate to members of a replica set.

- *Replica Set Tags*, 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 336), 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 217) and *sharding* (page 295), particularly *Replica Set Fundamental Concepts* (page 217) and *Sharded Cluster Overview* (page 295).

3.3 Journaling

MongoDB uses *write ahead logging* to an on-disk *journal* to guarantee *write operation* (page 37) 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 283) or, preferably, *resync* (page 231) from a clean member of the replica set.

With journaling enabled, if `mongod` stops unexpectedly, the program can recover everything written to the journal, and the data remains in a consistent state. By default, the greatest extent of lost writes, i.e., those not made to the journal, are those made in the last 100 milliseconds. See `journalCommitInterval` for more information on the default.

With journaling, if you want a data set to reside entirely in RAM, you need enough RAM to hold the dataset 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 103).

Important: Changed in version 2.0: For 64-bit builds of `mongod`, journaling is enabled by default. For other platforms, see `journal`.

3.3.1 Procedures

Enable Journaling

Changed in version 2.0: For 64-bit builds of `mongod`, journaling is enabled by default.

To enable journaling, start `mongod` with the `--journal` command line option.

If no journal files exist, when `mongod` starts, it must preallocate new journal files. During this operation, the `mongod` is not listening for connections until preallocation completes: for some systems this may take a several minutes. During this period your applications and the `mongo` shell are not available.

Disable Journaling

Warning: Do not disable journaling on production systems. If your `mongod` instance stops without shutting down cleanly unexpectedly for any reason, (e.g. power failure) and you are not running with journaling, then you must recover from an unaffected *replica set* member or backup, as described in *repair* (page 283).

To disable journaling, start `mongod` with the `--nojournal` command line option.

Get Commit Acknowledgment

You can get commit acknowledgment with the `getLastError` command and the `j` option. For details, see *Internal Operation of Write Concern* (page 39).

Avoid Preallocation Lag

To avoid *preallocation lag* (page 102), you can preallocate files in the journal directory by copying them from another instance of `mongod`.

Preallocated files do not contain data. It is safe to later remove them. But if you restart `mongod` with journaling, `mongod` will create them again.

Example

The following sequence preallocates journal files for an instance of `mongod` running on port `27017` with a database path of `/data/db`.

For demonstration purposes, the sequence starts by creating a set of journal files in the usual way.

1. Create a temporary directory into which to create a set of journal files:

   ```
   mkdir ~/tmpDbpath
   ```

2. Create a set of journal files by staring a `mongod` instance that uses the temporary directory:

   ```
   mongod --port 10000 --dbpath ~/tmpDbpath --journal
   ```

3. When you see the following log output, indicating `mongod` has the files, press CONTROL+C to stop the `mongod` instance:

   ```
   web admin interface listening on port 11000
   ```

4. Preallocate journal files for the new instance of `mongod` by moving the journal files from the data directory of the existing instance to the data directory of the new instance:

   ```
   mv ~/tmpDbpath/journal /data/db/
   ```

5. Start the new `mongod` instance:

   ```
   mongod --port 27017 --dbpath /data/db --journal
   ```

Monitor Journal Status

Use the following commands and methods to monitor journal status:

- `serverStatus`

  The `serverStatus` command returns database status information that is useful for assessing performance.

- `journalLatencyTest`

  Use `journalLatencyTest` to measure how long it takes on your volume to write to the disk in an append-only fashion. You can run this command on an idle system to get a baseline sync time for journaling. You can also run this command on a busy system to see the sync time on a busy system, which may be higher if the journal directory is on the same volume as the data files.

The `journalLatencyTest` command also provides a way to check if your disk drive is buffering writes in its local cache. If the number is very low (i.e., less than 2 milliseconds) and the drive is non-SSD, the drive is probably buffering writes. In that case, enable cache write-through for the device in your operating system, unless you have a disk controller card with battery backed RAM.

Change the Group Commit Interval

Changed in version 2.0.

You can set the group commit interval using the `--journalCommitInterval` command line option. The allowed range is `2` to `300` milliseconds.

Lower values increase the durability of the journal at the expense of disk performance.

Recover Data After Unexpected Shutdown

On a restart after a crash, MongoDB replays all journal files in the journal directory before the server becomes available. If MongoDB must replay journal files, `mongod` notes these events in the log output.

There is no reason to run `repairDatabase` in these situations.

3.3.2 Journaling Internals

When running with journaling, MongoDB stores and applies *write operations* (page 37) in memory and in the journal before the changes are in the data files.

Journal Files

With journaling enabled, MongoDB creates a journal directory within the directory defined by `dbpath`, which is `/data/db` by default. The journal directory holds journal files, which contain write-ahead redo logs. The directory also holds a last-sequence-number file. A clean shutdown removes all the files in the journal directory.

Journal files are append-only files and have file names prefixed with `j._`. When a journal file holds 1 gigabyte of data, MongoDB creates a new journal file. Once MongoDB applies all the write operations in the journal files, it deletes these files. Unless you write *many* bytes of data per-second, the journal directory should contain only two or three journal files.

To limit the size of each journal file to 128 megabytes, use the `smallfiles` run time option when starting `mongod`.

To speed the frequent sequential writes that occur to the current journal file, you can ensure that the journal directory is on a different system.

Important: If you place the journal on a different filesystem from your data files you *cannot* use a filesystem snapshot to capture consistent backups of a `dbpath` directory.

Note: Depending on your file system, you might experience a preallocation lag the first time you start a `mongod` instance with journaling enabled. MongoDB preallocates journal files if it is faster on your file system to create files of a pre-defined. 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 101).

Storage Views used in Journaling

Journaling adds three storage views to MongoDB.

The `shared view` stores modified data for upload to the MongoDB data files. The `shared view` is the only view with direct access to the MongoDB data files. When running with journaling, `mongod` asks the operating system to map your existing on-disk data files to the `shared view` memory view. The operating system maps the files but does not load them. MongoDB later loads data files to `shared view` as needed.

The `private view` stores data for use in *read operations* (page 25). MongoDB maps `private view` to the `shared view` and is the first place MongoDB applies new *write operations* (page 37).

The journal is an on-disk view that stores new write operations after MongoDB applies the operation to the `private cache` but before applying them to the data files. The journal provides durability. If the `mongod` instance were to crash without having applied the writes to the data files, the journal could replay the writes to the `shared view` for eventual upload to the data files.

How Journaling Records Write Operations

MongoDB copies the write operations to the journal in batches called group commits. See `journalCommitInterval` for more information on the default commit interval. These "group commits" help minimize the performance impact of journaling.

Journaling stores raw operations that allow MongoDB to reconstruct the following:

- document insertion/updates
- index modifications
- changes to the namespace files

As *write operations* (page 37) occur, MongoDB writes the data to the `private view` in RAM and then copies the write operations in batches to the journal. The journal stores the operations on disk to ensure durability. MongoDB adds the operations as entries on the journal's forward pointer. Each entry describes which bytes the write operation changed in the data files.

MongoDB next applies the journal's write operations to the `shared view`. At this point, the `shared view` becomes inconsistent with the data files.

At default intervals of 60 seconds, MongoDB asks the operating system to flush the `shared view` to disk. This brings the data files up-to-date with the latest write operations.

When MongoDB flushes write operations to the data files, MongoDB removes the write operations from the journal's behind pointer. The behind pointer is always far back from advanced pointer.

As part of journaling, MongoDB routinely asks the operating system to remap the `shared view` to the `private view`, for consistency.

Note: The interaction between the `shared view` and the on-disk data files is similar to how MongoDB works *without* journaling, which is that MongoDB asks the operating system to flush in-memory changes back to the data files every 60 seconds.

3.4 Use MongoDB with SSL Connections

This document outlines the use and operation of MongoDB's SSL support. SSL allows MongoDB clients to support encrypted connections to `mongod` instances.

Note: The default distribution of MongoDB[3] does **not** contain support for SSL.

As of the current release, to use SSL you must either: build MongoDB locally passing the "`--ssl`" option to `scons`, or use the MongoDB subscriber build[4].

These instructions outline the process for getting started with SSL and assume that you have already installed a build of MongoDB that includes SSL support and that your client driver supports SSL.

3.4.1 `mongod` and `mongos` SSL Configuration

Add the following command line options to your `mongod` invocation:

```
mongod --sslOnNormalPorts --sslPEMKeyFile <pem> --sslPEMKeyPassword <pass>
```

Replace "`<pem>`" with the path to your SSL certificate `.pem` file, and "`<pass>`" with the password you used to encrypt the `.pem` file.

You may also specify these options in your "`mongodb.conf`" file, as in the following:

```
sslOnNormalPorts = true
sslPEMKeyFile = /etc/ssl/mongodb.pem
sslPEMKeyPassword = pass
```

Modify these values to reflect the location of your actual `.pem` file and its password.

You can specify these configuration options in a configuration file for `mongos`, or start `mongos` with the following invocation:

```
mongos --sslOnNormalPorts --sslPEMKeyFile <pem> --sslPEMKeyPassword <pass>
```

You can use any existing SSL certificate, or you can generate your own SSL certificate using a command that resembles the following:

```
cd /etc/ssl/
openssl req -new -x509 -days 365 -nodes -out mongodb-cert.pem -keyout mongodb-cert.key
```

To create the combined `.pem` file that contains the `.key` file and the `.pem` certificate, use the following command:

```
cat mongodb-cert.key mongodb-cert.pem > mongodb.pem
```

3.4.2 Clients

Clients must have support for SSL to work with a `mongod` instance that has SSL support enabled. The current versions of the Python, Java, Ruby, Node.js, and .NET drivers have support for SSL, with full support coming in future releases of other drivers.

`mongo`

The `mongo` shell built with ssl support distributed with the subscriber build also supports SSL. Use the "`--ssl`" flag as follows:

```
mongo --ssl --host <host>
```

[3] http://www.mongodb.org/downloads
[4] http://www.mongodb.com/mongodb-subscriber-edition-download

MMS

The MMS agent will also have to connect via SSL in order to gather its stats. Because the agent already utilizes SSL for its communications to the MMS servers, this is just a matter of enabling SSL support in MMS itself on a per host basis.

Please see the MMS Manual[5] for more information about MMS configuration.

PyMongo

Add the "`ssl=True`" parameter to a PyMongo `MongoClient`[6] to create a MongoDB connection to an SSL MongoDB instance:

```
from pymongo import MongoClient
c = MongoClient(host="mongodb.example.net", port=27017, ssl=True)
```

To connect to a replica set, use the following operation:

```
from pymongo import MongoReplicaSetClient
c = MongoReplicaSetClient("mongodb.example.net:27017",
                          replicaSet="mysetname", ssl=True)
```

PyMongo also supports an "`ssl=true`" option for the MongoDB URI:

```
mongodb://mongodb.example.net:27017/?ssl=true
```

Java

Consider the following example "`SSLApp.java`" class file:

```
import com.mongodb.*;
import javax.net.ssl.SSLSocketFactory;

public class SSLApp {

    public static void main(String args[])  throws Exception {

        MongoClientOptions o = new MongoClientOptions.Builder()
                .socketFactory(SSLSocketFactory.getDefault())
                .build();

        MongoClient m = new MongoClient("localhost", o);

        DB db = m.getDB( "test" );
        DBCollection c = db.getCollection( "foo" );

        System.out.println( c.findOne() );
    }
}
```

Ruby

The recent versions of the Ruby driver have support for connections to SSL servers. Install the latest version of the driver with the following command:

[5] http://mms.mongodb.com/help
[6] http://api.mongodb.org/python/current/api/pymongo/mongo_client.html#pymongo.mongo_client.MongoClient

```
gem install mongo
```

Then connect to a standalone instance, using the following form:

```
require 'rubygems'
require 'mongo'

connection = Mongo::Connection.new('localhost', 27017, :ssl => true)
```

Replace `connection` with the following if you're connecting to a replica set:

```
connection = Mongo::ReplSetConnection.new(['localhost:27017'],
                                          ['localhost:27018'],
                                          :ssl => true)
```

Here, `mongod` instance run on "`localhost:27017`" and "`localhost:27018`".

Node.JS (`node-mongodb-native`)

In the node-mongodb-native[7] driver, use the following invocation to connect to a `mongod` or `mongos` instance via SSL:

```
var db1 = new Db(MONGODB, new Server("127.0.0.1", 27017,
                                     { auto_reconnect: false, poolSize:4, ssl:ssl } );
```

To connect to a replica set via SSL, use the following form:

```
var replSet = new ReplSetServers( [
    new Server( RS.host, RS.ports[1], { auto_reconnect: true } ),
    new Server( RS.host, RS.ports[0], { auto_reconnect: true } ),
    ],
  {rs_name:RS.name, ssl:ssl}
);
```

.NET

As of release 1.6, the .NET driver supports SSL connections with `mongod` and `mongos` instances. To connect using SSL, you must add an option to the connection string, specifying `ssl=true` as follows:

```
var connectionString = "mongodb://localhost/?ssl=true";
var server = MongoServer.Create(connectionString);
```

The .NET driver will validate the certificate against the local trusted certificate store, in addition to providing encryption of the server. This behavior may produce issues during testing, if the server uses a self-signed certificate. If you encounter this issue, add the `sslverifycertificate=false` option to the connection string to prevent the .NET driver from validating the certificate, as follows:

```
var connectionString = "mongodb://localhost/?ssl=true&sslverifycertificate=false";
var server = MongoServer.Create(connectionString);
```

3.5 Use MongoDB with SNMP Monitoring

New in version 2.2.

[7] https://github.com/mongodb/node-mongodb-native

Subscriber Only Feature

This feature is only available in the Subscriber Edition of MongoDB.

This document outlines the use and operation of MongoDB's SNMP extension, which is is only available in the MongoDB Subscriber Edition[8].

3.5.1 Prerequisites

Install MongoDB Subscriber Edition

The MongoDB Subscriber Edition, is available on four platforms. For more information, see MongoDB Subscriber Edition[9].

Included Files

The Subscriber Edition packages contain the following files:

- `MONGO-MIB.txt`:

  The `MIB` file that describes the data (i.e. schema) for MongoDB's SNMP output

- `mongod.conf`:

  The SNMP configuration file for reading the SNMP output of MongoDB. The SNMP configures the community names, permissions, access controls, etc.

Required Packages

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

- Ubuntu 11.04 requires `libssl0.9.8`, `snmp-mibs-downloader`, `snmp`, and `snmpd`. Issue a command such as the following to install these packages:

  ```
  sudo apt-get install libssl0.9.8 snmp snmpd snmp-mibs-downloader
  ```

- Red Hat Enterprise Linux 6.x series and Amazon Linux AMI require `libssl`, `net-snmp`, `net-snmp-libs`, and `net-snmp-utils`. Issue a command such as the following to install these packages:

  ```
  sudo yum install libssl net-snmp net-snmp-libs net-snmp-utils
  ```

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

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

[8] http://www.mongodb.com/mongodb-subscriber-edition-download
[9] http://www.mongodb.com/mongodb-subscriber-edition-download

3.5.2 Configure SNMP

Install MIB Configuration Files

Ensure that the MIB directory, at `/usr/share/snmp/mibs` exists. If not, issue the following command:

```
sudo mkdir -p /usr/share/snmp/mibs
```

Use the following command to create a symbolic link:

```
sudo ln -s [/path/to/mongodb/distribution/]MONGO-MIB.txt /usr/share/snmp/mibs/
```

Replace `[/path/to/mongodb/distribution/]` with the path to your `MONGO-MIB.txt` configuration file.

Copy the `mongod.conf` file into the `/etc/snmp` directory with the following command:

```
cp mongod.conf /etc/snmp/mongod.conf
```

Start Up

You can control the Subscriber Edition of MongoDB using default or custom or control scripts, just as you can any other **mongod**:

Use the following command to view all SNMP options available in your MongoDB:

```
mongod --help | grep snmp
```

The above command should return the following output:

```
Module snmp options:
  --snmp-subagent       run snmp subagent
  --snmp-master         run snmp as master
```

Ensure that the following directories exist:

- `/data/db/` (This is the path where MongoDB stores the data files.)
- `/var/log/mongodb/` (This is the path where MongoDB writes the log output.)

If they do not, issue the following command:

```
mkdir -p /var/log/mongodb/ /data/db/
```

Start the **mongod** instance with the following command:

```
mongod --snmp-master --port 3001 --fork --dbpath /data/db/  --logpath /var/log/mongodb/1.log
```

Optionally, you can set these options in a `configuration file`.

To check if **mongod** is running with SNMP support, issue the following command:

```
ps -ef | grep 'mongod --snmp'
```

The command should return output that includes the following line. This indicates that the proper **mongod** instance is running:

```
systemuser 31415 10260  0 Jul13 pts/16   00:00:00 mongod --snmp-master --port 3001 # [...]
```

Test SNMP

Check for the snmp agent process listening on port `1161` with the following command:

```
sudo lsof -i :1161
```

which return the following output:

```
COMMAND  PID     USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
mongod  9238 sysadmin   10u  IPv4  96469      0t0  UDP localhost:health-polling
```

Similarly, this command:

```
netstat -an | grep 1161
```

should return the following output:

```
udp        0      0 127.0.0.1:1161              0.0.0.0:*
```

Run `snmpwalk` Locally

`snmpwalk` provides tools for retrieving and parsing the SNMP data according to the MIB. If you installed all of the required packages above, your system will have `snmpwalk`.

Issue the following command to collect data from **mongod** using SNMP:

```
snmpwalk -m MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

You may also choose to specify the path to the MIB file:

```
snmpwalk -m /usr/share/snmp/mibs/MONGO-MIB -v 2c -c mongodb 127.0.0.1:1161 1.3.6.1.4.1.37601
```

Use this command *only* to ensure that you can retrieve and validate SNMP data from MongoDB.

3.5.3 Troubleshooting

Always check the logs for errors if something does not run as expected, see the log at `/var/log/mongodb/1.log`. The presence of the following line indicates that the **mongod** cannot read the `/etc/snmp/mongod.conf` file:

```
[SNMPAgent] warning: error starting SNMPAgent as master err:1
```

3.6 Monitoring Database Systems

Monitoring is a critical component of all database administration. A firm grasp of MongoDB's reporting will allow you to assess the state of your database and maintain your deployment without crisis. Additionally, a sense of MongoDB's normal operational parameters will allow you to diagnose issues as you encounter them, rather than waiting for a crisis or failure.

This document provides an overview of the available tools and data provided by MongoDB as well as an introduction to diagnostic strategies, and suggestions for monitoring instances in MongoDB's replica sets and sharded clusters.

Note: MMS (MongoDB Management Service)[10] is a hosted monitoring service which collects and aggregates data

to provide insight into the performance and operation of MongoDB deployments. See the MMS documentation[11] for more information.

3.6.1 Monitoring Tools

There are two primary methods for collecting data regarding the state of a running MongoDB instance. First, there are a set of tools distributed with MongoDB that provide real-time reporting of activity on the database. Second, several `database commands` return statistics regarding the current database state with greater fidelity. Both methods allow you to collect data that answers a different set of questions, and are useful in different contexts.

This section provides an overview of these utilities and statistics, along with an example of the kinds of questions that each method is most suited to help you address.

Utilities

The MongoDB distribution includes a number of utilities that return statistics about instances' performance and activity quickly. These are typically most useful for diagnosing issues and assessing normal operation.

mongotop

`mongotop` tracks and reports the current read and write activity of a MongoDB instance. `mongotop` provides per-collection visibility into use. Use `mongotop` to verify that activity and use match expectations. See the `mongotop manual` for details.

mongostat

`mongostat` captures and returns counters of database operations. `mongostat` reports operations on a per-type (e.g. insert, query, update, delete, etc.) basis. This format makes it easy to understand the distribution of load on the server. Use `mongostat` to understand the distribution of operation types and to inform capacity planning. See the `mongostat manual` for details.

REST Interface

MongoDB provides a *REST* interface that exposes a diagnostic and monitoring information in a simple web page. Enable this by setting `rest` to `true`, and access this page via the local host interface using the port numbered 1000 more than that the database port. In default configurations the REST interface is accessible on `28017`. For example, to access the REST interface on a locally running mongod instance: http://localhost:28017

Statistics

MongoDB provides a number of commands that return statistics about the state of the MongoDB instance. These data may provide finer granularity regarding the state of the MongoDB instance than the tools above. Consider using their output in scripts and programs to develop custom alerts, or to modify the behavior of your application in response to the activity of your instance.

[10] http://mms.mongodb.com
[11] http://mms.mongodb.com/help/

serverStatus

Access `serverStatus data` by way of the `serverStatus` command. This *document* contains a general overview of the state of the database, including disk usage, memory use, connection, journaling, index accesses. The command returns quickly and does not impact MongoDB performance.

While this output contains a (nearly) complete account of the state of a MongoDB instance, in most cases you will not run this command directly. Nevertheless, all administrators should be familiar with the data provided by `serverStatus`.

See also:

`db.serverStatus()` and `serverStatus data`.

replSetGetStatus

View the `replSetGetStatus data` with the `replSetGetStatus` (page 291) command (`rs.status()` (page 286) from the shell). The document returned by this command reflects the state and configuration of the replica set. Use this data to ensure that replication is properly configured, and to check the connections between the current host and the members of the replica set.

dbStats

The `dbStats data` is accessible by way of the `dbStats` command (`db.stats()` from the shell). This command returns a document that contains data that reflects the amount of storage used and data contained in the database, as well as object, collection, and index counters. Use this data to check and track the state and storage of a specific database. This output also allows you to compare utilization between databases and to determine average *document* size in a database.

collStats

The `collStats data` is accessible using the `collStats` command (`db.printCollectionStats()` from the shell). It provides statistics that resemble `dbStats` on the collection level: this includes a count of the objects in the collection, the size of the collection, the amount of disk space used by the collection, and information about the indexes.

Introspection Tools

In addition to status reporting, MongoDB provides a number of introspection tools that you can use to diagnose and analyze performance and operational conditions. Consider the following documentation:

- `diagLogging`
- `http://docs.mongodb.org/manual/tutorial/manage-the-database-profiler`
- `http://docs.mongodb.org/manual/reference/database-profiler`
- `http://docs.mongodb.org/manual/reference/current-op`

Third Party Tools

A number of third party monitoring tools have support for MongoDB, either directly, or through their own plugins.

Self Hosted Monitoring Tools

These are monitoring tools that you must install, configure and maintain on your own servers, usually open source.

Tool	Plugin	Description
Ganglia[12]	mongodb-ganglia[13]	Python script to report operations per second, memory usage, btree statistics, master/slave status and current connections.
Ganglia	gmond_python_modules[14]	Parses output from the `serverStatus` and `replSetGetStatus` (page 291) commands.
Motop[15]	*None*	Realtime monitoring tool for several MongoDB servers. Shows current operations ordered by durations every second.
mtop[16]	*None*	A top like tool.
Munin[17]	mongo-munin[18]	Retrieves server statistics.
Munin	mongomon[19]	Retrieves collection statistics (sizes, index sizes, and each (configured) collection count for one DB).
Munin	munin-plugins Ubuntu PPA[20]	Some additional munin plugins not in the main distribution.
Nagios[21]	nagios-plugin-mongodb[22]	A simple Nagios check script, written in Python.

Also consider dex[23], an index and query analyzing tool for MongoDB that compares MongoDB log files and indexes to make indexing recommendations.

Hosted (SaaS) Monitoring Tools

These are monitoring tools provided as a hosted service, usually on a subscription billing basis.

Name	Notes
MongoDB Management Service[24]	**MMS**[25] is a cloud-based suite of services for managing MongoDB deployments. MMS provides monitoring and backup capabilities.
Scout[26]	Several plugins including: MongoDB Monitoring[27], MongoDB Slow Queries[28] and MongoDB Replica Set Monitoring[29].
Server Density[30]	Dashboard for MongoDB[31], MongoDB specific alerts, replication failover timeline and iPhone, iPad and Android mobile apps.

[12] http://sourceforge.net/apps/trac/ganglia/wiki
[13] https://github.com/quiiver/mongodb-ganglia
[14] https://github.com/ganglia/gmond_python_modules
[15] https://github.com/tart/motop
[16] https://github.com/beaufour/mtop
[17] http://munin-monitoring.org/
[18] https://github.com/erh/mongo-munin
[19] https://github.com/pcdummy/mongomon
[20] https://launchpad.net/ chris-lea/+archive/munin-plugins
[21] http://www.nagios.org/
[22] https://github.com/mzupan/nagios-plugin-mongodb
[23] https://github.com/mongolab/dex

3.6.2 Process Logging

During normal operation, `mongod` and `mongos` instances report information that reflect current operation to standard output, or a log file. The following runtime settings control these options.

- `quiet`. Limits the amount of information written to the log or output.
- `verbose`. Increases the amount of information written to the log or output.

  You can also specify this as `v` (as in `-v`.) Set multiple `v`, as in `vvvv = True` for higher levels of verbosity. You can also change the verbosity of a running `mongod` or `mongos` instance with the `setParameter` command.
- `logpath`. Enables logging to a file, rather than standard output. Specify the full path to the log file to 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`

Additionally, the following *database commands* affect logging:

- `getLog`. Displays recent messages from the `mongod` process log.
- `logRotate`. Rotates the log files for `mongod` processes only. See `http://docs.mongodb.org/manual/tutorial/rotate-log-files`.

3.6.3 Diagnosing Performance Issues

Degraded performance in MongoDB can be the result of an array of causes, and is typically a function of the relationship among the quantity of data stored in the database, the amount of system RAM, the number of connections to the database, and the amount of time the database spends in a lock state.

In some cases performance issues may be transient and related to traffic load, data access patterns, or the availability of hardware on the host system for virtualized environments. Some users also experience performance limitations as a result of inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns. In other situations, performance issues may indicate that the database may be operating at capacity and that it is time to add additional capacity to the database.

Locks

MongoDB uses a locking system to ensure consistency. However, if certain operations are long-running, or a queue forms, performance slows as requests and operations wait for the lock. Because lock related slow downs can be intermittent, look to the data in the *globalLock* section of the `serverStatus` response to assess if the lock has been a challenge to your performance. If `globalLock.currentQueue.total` is consistently high, then there is a chance that a large number of requests are waiting for a lock. This indicates a possible concurrency issue that might affect performance.

If `globalLock.totalTime` is high in context of `uptime` then the database has existed in a lock state for a significant amount of time. If `globalLock.ratio` is also high, MongoDB has likely been processing a large number of long running queries. Long queries are often the result of a number of factors: ineffective use of indexes, non-optimal schema design, poor query structure, system architecture issues, or insufficient RAM resulting in *page faults* (page 114) and disk reads.

Memory Usage

Because MongoDB uses memory mapped files to store data, given a data set of sufficient size, the MongoDB process will allocate all memory available on the system for its use. Because of the way operating systems function, the amount of allocated RAM is not a useful reflection of MongoDB's state.

While this is part of the design, and affords MongoDB superior performance, the memory mapped files make it difficult to determine if the amount of RAM is sufficient for the data set. Consider *memory usage statuses* to better understand MongoDB's memory utilization. Check the resident memory use (i.e. `mem.resident`:) if this exceeds the amount of system memory *and* there's a significant amount of data on disk that isn't in RAM, you may have exceeded the capacity of your system.

Also check the amount of mapped memory (i.e. `mem.mapped`.) If this value is greater than the amount of system memory, some operations will require disk access *page faults* to read data from virtual memory with deleterious effects on performance.

Page Faults

Page faults represent the number of times that MongoDB requires data not located in physical memory, and must read from virtual memory. To check for page faults, see the `extra_info.page_faults` value in the `serverStatus` command. This data is only available on Linux systems.

Alone, page faults are minor and complete quickly; however, in aggregate, large numbers of page fault typically indicate that MongoDB is reading too much data from disk and can indicate a number of underlying causes and recommendations. In many situations, MongoDB's read locks will "yield" after a page fault to allow other processes to read and avoid blocking while waiting for the next page to read into memory. This approach improves concurrency, and in high volume systems this also improves overall throughput.

If possible, increasing the amount of RAM accessible to MongoDB may help reduce the number of page faults. If this is not possible, you may want to consider deploying a *sharded cluster* and/or adding one or more *shards* to your deployment to distribute load among `mongod` instances.

Number of Connections

In some cases, the number of connections between the application layer (i.e. clients) and the database can overwhelm the ability of the server to handle requests which can produce performance irregularities. Check the following fields in the `serverStatus` document:

- `globalLock.activeClients` contains a counter of the total number of clients with active operations in progress or queued.
- `connections` is a container for the following two fields:
  - `current` the total number of current clients that connect to the database instance.
  - `available` the total number of unused collections available for new clients.

Note: Unless limited by system-wide limits MongoDB has a hard connection limit of 20 thousand connections. You can modify system limits using the `ulimit` command, or by editing your system's `/etc/sysctl` file.

If requests are high because there are many concurrent application requests, the database may have trouble keeping up with demand. If this is the case, then you will need to increase the capacity of your deployment. For read-heavy applications increase the size of your *replica set* and distribute read operations to *secondary* members. For write heavy applications, deploy *sharding* and add one or more *shards* to a *sharded cluster* to distribute load among `mongod` instances.

Spikes in the number of connections can also be the result of application or driver errors. All of the officially supported MongoDB drivers implement connection pooling, which allows clients to use and reuse connections more efficiently. Extremely high numbers of connections, particularly without corresponding workload is often indicative of a driver or other configuration error.

Database Profiling

MongoDB contains a database profiling system that can help identify inefficient queries and operations. Enable the profiler by setting the `profile` value using the following command in the `mongo` shell:

```
db.setProfilingLevel(1)
```

See

The documentation of `db.setProfilingLevel()` for more information about this command.

Note: Because the database profiler can have an impact on the performance, only enable profiling for strategic intervals and as minimally as possible on production systems.

You may enable profiling on a per-`mongod` basis. This setting will not propagate across a *replica set* or *sharded cluster*.

The following profiling levels are available:

Level	Setting
0	Off. No profiling.
1	On. Only includes slow operations.
2	On. Includes all operations.

See the output of the profiler in the `system.profile` collection of your database. You can specify the `slowms` setting to set a threshold above which the profiler considers operations "slow" and thus included in the level `1` profiling data. You may configure `slowms` at runtime, as an argument to the `db.setProfilingLevel()` operation.

Additionally, `mongod` records all "slow" queries to its `log`, as defined by `slowms`. The data in `system.profile` does not persist between `mongod` restarts.

You can view the profiler's output by issuing the `show profile` command in the `mongo` shell, with the following operation.

```
db.system.profile.find( { millis : { $gt : 100 } } )
```

This returns all operations that lasted longer than 100 milliseconds. Ensure that the value specified here (i.e. `100`) is above the `slowms` threshold.

See also:

`http://docs.mongodb.org/manual/applications/optimization` addresses strategies that may improve the performance of your database queries and operations.

3.6.4 Replication and Monitoring

The primary administrative concern that requires monitoring with replica sets, beyond the requirements for any MongoDB instance, is "replication lag." This refers to the amount of time that it takes a write operation on the *primary* to replicate to a *secondary*. Some very small delay period may be acceptable; however, as replication lag grows, two significant problems emerge:

- First, operations that have occurred in the period of lag are not replicated to one or more secondaries. If you're using replication to ensure data persistence, exceptionally long delays may impact the integrity of your data set.
- Second, if the replication lag exceeds the length of the operation log (*oplog*) then MongoDB will have to perform an initial sync on the secondary, copying all data from the *primary* and rebuilding all indexes. In normal circumstances this is uncommon given the typical size of the oplog, but it's an issue to be aware of.

For causes of replication lag, see *Replication Lag* (page 233).

Replication issues are most often the result of network connectivity issues between members or the result of a *primary* that does not have the resources to support application and replication traffic. To check the status of a replica, use the `replSetGetStatus` (page 291) or the following helper in the shell:

```
rs.status()
```

See the `http://docs.mongodb.org/manual/reference/replica-status` document for a more in depth overview view of this output. In general watch the value of `optimeDate`. Pay particular attention to the difference in time between the *primary* and the *secondary* members.

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

By default the oplog is 5% of total available disk space on 64-bit systems.

See also:

Change the Size of the Oplog (page 271)

3.6.5 Sharding and Monitoring

In most cases the components of *sharded clusters* benefit from the same monitoring and analysis as all other MongoDB instances. Additionally, clusters require monitoring to ensure that data is effectively distributed among nodes and that sharding operations are functioning appropriately.

See also:

See the *Sharding* (page 295) page for more information.

Config Servers

The *config database* provides a map of documents to shards. The cluster updates this map as *chunks* move between shards. When a configuration server becomes inaccessible, some sharding operations like moving chunks and starting `mongos` instances become unavailable. However, clusters remain accessible from already-running `mongos` instances.

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

Balancing and Chunk Distribution

The most effective *sharded cluster* deployments require that *chunks* are evenly balanced among the shards. MongoDB has a background *balancer* process that distributes data such that chunks are always optimally distributed among the *shards*. Issue the `db.printShardingStatus()` or `sh.status()` (page 349) command to the `mongos` by way of the `mongo` shell. This returns an overview of the entire cluster including the database name, and a list of the chunks.

Stale Locks

In nearly every case, all locks used by the balancer are automatically released when they become stale. However, because any long lasting lock can block future balancing, it's important to insure that all locks are legitimate. To check the lock status of the database, connect to a `mongos` instance using the `mongo` shell. Issue the following command sequence to switch to the `config` database and display all outstanding locks on the shard database:

```
use config
db.locks.find()
```

For active deployments, the above query might return a useful result set. 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.

3.7 Importing and Exporting MongoDB Data

Full *database instance backups* (page 120) are useful for disaster recovery protection and routine database backup operation; however, some cases require additional import and export functionality.

This document provides an overview of the import and export programs included in the MongoDB distribution. These tools are useful when you want to backup or export a portion of your data without capturing the state of the entire database, or for simple data ingestion cases. For more complex data migration tasks, you may want to write your own import and export scripts using a client *driver* to interact with the database itself.

Warning: Because these tools primarily operate by interacting with a running `mongod` instance, they can impact the performance of your running database.
Not only do these processes create traffic for a running database instance, they also force the database to read all data through memory. When MongoDB reads infrequently used data, it can supplant more frequently accessed data, causing a deterioration in performance for the database's regular workload.
`mongoimport` and `mongoexport` do not reliably preserve all rich *BSON* data types, because *BSON* is a superset of *JSON*. Thus, `mongoimport` and `mongoexport` cannot represent *BSON* data accurately in *JSON*. As a result data exported or imported with these tools may lose some measure of fidelity. See `http://docs.mongodb.org/manual/reference/mongodb-extended-json` for more information about MongoDB Extended JSON.

See also:

See the "*Backup Strategies for MongoDB Systems* (page 120)" document for more information on backing up MongoDB instances. Additionally, consider the following references for commands addressed in this document:

- `http://docs.mongodb.org/manual/reference/mongoexport`
- `http://docs.mongodb.org/manual/reference/mongorestore`
- `http://docs.mongodb.org/manual/reference/mongodump`

If you want to transform and process data once you've imported it in MongoDB consider the topics in *Aggregation* (page 151), including:

- *Map-Reduce* (page 174) and
- *Aggregation Framework* (page 151).

3.7.1 Data Type Fidelity

JSON does not have the following data types that exist in *BSON* documents: `data_binary`, `data_date`, `data_timestamp`, `data_regex`, `data_oid` and `data_ref`. As a result using any tool that decodes BSON *documents* into JSON will suffer some loss of fidelity.

If maintaining type fidelity is important, consider writing a data import and export system that does not force BSON documents into JSON form as part of the process. The following list of types contain examples for how MongoDB will represent how BSON documents render in JSON.

- `data_binary`

  ```
  { "$binary" : "<bindata>", "$type" : "<t>" }
  ```

  `<bindata>` is the base64 representation of a binary string. `<t>` is the hexadecimal representation of a single byte indicating the data type.

- `data_date`

  ```
  Date( <date> )
  ```

  `<date>` is the JSON representation of a 64-bit signed integer for milliseconds since epoch.

- `data_timestamp`

  ```
  Timestamp( <t>, <i> )
  ```

  `<t>` is the JSON representation of a 32-bit unsigned integer for milliseconds since epoch. `<i>` is a 32-bit unsigned integer for the increment.

- `data_regex`

  ```
  /<jRegex>/<jOptions>
  ```

  `<jRegex>` is a string that may contain valid JSON characters and unescaped double quote (i.e. `"`) characters, but may not contain unescaped forward slash (i.e. `http://docs.mongodb.org/manual/`) characters. `<jOptions>` is a string that may contain only the characters `g`, `i`, `m`, and `s`.

- `data_oid`

  ```
  ObjectId( "<id>" )
  ```

  `<id>` is a 24 character hexadecimal string. These representations require that `data_oid` values have an associated field named "*_id*."

- `data_ref`

  ```
  DBRef( "<name>", "<id>" )
  ```

  `<name>` is a string of valid JSON characters. `<id>` is a 24 character hexadecimal string.

See also:

`MongoDB Extended JSON`

3.7.2 Data Import and Export and Backups Operations

For resilient and non-disruptive backups, use a file system or block-level disk snapshot function, such as the methods described in the "*Backup Strategies for MongoDB Systems* (page 120)" document. The tools and operations discussed provide functionality that's useful in the context of providing some kinds of backups.

By contrast, use import and export tools to backup a small subset of your data or to move data to or from a 3rd party system. These backups may capture a small crucial set of data or a frequently modified section of data, for extra insurance, or for ease of access. No matter how you decide to import or export your data, consider the following guidelines:

- Label files so that you can identify what point in time the export or backup reflects.
- Labeling should describe the contents of the backup, and reflect the subset of the data corpus, captured in the backup or export.
- Do not create or apply exports if the backup process itself will have an adverse effect on a production system.
- Make sure that they reflect a consistent data state. Export or backup processes can impact data integrity (i.e. type fidelity) and consistency if updates continue during the backup process.
- Test backups and exports by restoring and importing to ensure that the backups are useful.

3.7.3 Human Intelligible Import/Export Formats

This section describes a process to import/export your database, or a portion thereof, to a file in a *JSON* or *CSV* format.

See also:

The `http://docs.mongodb.org/manual/reference/mongoimport` and `http://docs.mongodb.org/manual/reference/mongoexport` documents contain complete documentation of these tools. If you have questions about the function and parameters of these tools not covered here, please refer to these documents.

If you want to simply copy a database or collection from one instance to another, consider using the `copydb`, `clone`, or `cloneCollection` commands, which may be more suited to this task. The `mongo` shell provides the `db.copyDatabase()` method.

These tools may also be useful for importing data into a MongoDB database from third party applications.

Collection Export with `mongoexport`

With the `mongoexport` utility you can create a backup file. In the most simple invocation, the command takes the following form:

```
mongoexport --collection collection --out collection.json
```

This will export all documents in the collection named `collection` into the file `collection.json`. Without the output specification (i.e. "*`--out collection.json`*",) `mongoexport` writes output to standard output (i.e. "stdout.") You can further narrow the results by supplying a query filter using the "*`--query`*" and limit results to a single database using the "*`--db`*" option. For instance:

```
mongoexport --db sales --collection contacts --query '{"field": 1}'
```

This command returns all documents in the `sales` database's `contacts` collection, with a field named `field` with a value of `1`. Enclose the query in single quotes (e.g. `'`) to ensure that it does not interact with your shell environment. The resulting documents will return on standard output.

By default, `mongoexport` returns one *JSON document* per MongoDB document. Specify the "*`--jsonArray`*" argument to return the export as a single *JSON* array. Use the "*`--csv`*" file to return the result in CSV (comma separated values) format.

If your `mongod` instance is not running, you can use the "*`--dbpath`*" option to specify the location to your MongoDB instance's database files. See the following example:

```
mongoexport --db sales --collection contacts --dbpath /srv/MongoDB/
```

This reads the data files directly. This locks the data directory to prevent conflicting writes. The `mongod` process must *not* be running or attached to these data files when you run `mongoexport` in this configuration.

The "`--host`" and "`--port`" options allow you to specify a non-local host to connect to capture the export. Consider the following example:

```
mongoexport --host mongodb1.example.net --port 37017 --username user --password pass --collection co
```

On any `mongoexport` command you may, as above specify username and password credentials as above.

Collection Import with `mongoimport`

To restore a backup taken with `mongoexport`. Most of the arguments to `mongoexport` also exist for `mongoimport`. Consider the following command:

```
mongoimport --collection collection --file collection.json
```

This imports the contents of the file `collection.json` into the collection named `collection`. If you do not specify a file with the "`--file`" option, `mongoimport` accepts input over standard input (e.g. "stdin.")

If you specify the "`--upsert`" option, all of `mongoimport` operations will attempt to update existing documents in the database and insert other documents. This option will cause some performance impact depending on your configuration.

You can specify the database option `--db` to import these documents to a particular database. If your MongoDB instance is not running, use the "`--dbpath`" option to specify the location of your MongoDB instance's database files. Consider using the "`--journal`" option to ensure that `mongoimport` records its operations in the journal. The `mongod` process must *not* be running or attached to these data files when you run `mongoimport` in this configuration.

Use the "`--ignoreBlanks`" option to ignore blank fields. For *CSV* and *TSV* imports, this option provides the desired functionality in most cases: it avoids inserting blank fields in MongoDB documents.

3.8 Backup Strategies for MongoDB Systems

Backups are an important part of any operational disaster recovery plan. A good backup plan must be able to capture data in a consistent and usable state, and operators must be able to automate both the backup and the recovery operations. Also test all components of the backup system to ensure that you can recover backed up data as needed. If you cannot effectively restore your database from the backup, then your backups are useless. This document addresses higher level backup strategies, for more information on specific backup procedures consider the following documents:

- `http://docs.mongodb.org/manual/tutorial/backup-databases-with-filesystem-snapshots`.
- `http://docs.mongodb.org/manual/tutorial/backup-databases-with-binary-database-dumps`.
- *Backup a Small Sharded Cluster with mongodump* (page 331)
- *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 332)
- *Create Backup of a Sharded Cluster with Database Dumps* (page 333)
- *Schedule Backup Window for Sharded Clusters* (page 336)
- *Restore a Single Shard* (page 334)
- *Restore Sharded Clusters* (page 335)

3.8.1 Backup Considerations

As you develop a backup strategy for your MongoDB deployment consider the following factors:

- Geography. Ensure that you move some backups away from the your primary database infrastructure.
- System errors. Ensure that your backups can survive situations where hardware failures or disk errors impact the integrity or availability of your backups.
- Production constraints. Backup operations themselves sometimes require substantial system resources. It is important to consider the time of the backup schedule relative to peak usage and maintenance windows.
- System capabilities. Some of the block-level snapshot tools require special support on the operating-system or infrastructure level.
- Database configuration. *Replication* and *sharding* can affect the process and impact of the backup implementation. See *Sharded Cluster Backup Considerations* (page 121) and *Replica Set Backup Considerations* (page 122).
- Actual requirements. You may be able to save time, effort, and space by including only crucial data in the most frequent backups and backing up less crucial data less frequently.

3.8.2 Approaches to Backing Up MongoDB Systems

There are two main methodologies for backing up MongoDB instances. Creating binary "dumps" of the database using `mongodump` or creating filesystem level snapshots. Both methodologies have advantages and disadvantages:

- binary database dumps are comparatively small, because they don't include index content or pre-allocated free space, and *record padding* (page 41). However, it's impossible to capture a copy of a running system that reflects a single moment in time using a binary dump.
- filesystem snapshots, sometimes called block level backups, produce larger backup sizes, but complete quickly and can reflect a single moment in time on a running system. However, snapshot systems require filesystem and operating system support and tools.

The best option depends on the requirements of your deployment and disaster recovery needs. Typically, filesystem snapshots are because of their accuracy and simplicity; however, `mongodump` is a viable option used often to generate backups of MongoDB systems.

The following topics provide details and procedures on the two approaches:

- `http://docs.mongodb.org/manual/tutorial/backup-databases-with-filesystem-snapshots`.
- `http://docs.mongodb.org/manual/tutorial/backup-databases-with-binary-database-dumps`.

In some cases, taking backups is difficult or impossible because of large data volumes, distributed architectures, and data transmission speeds. In these situations, increase the number of members in your replica set or sets.

3.8.3 Backup Strategies for MongoDB Deployments

Sharded Cluster Backup Considerations

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Sharded clusters complicate backup operations, as distributed systems. True point-in-time backups are only possible when stopping all write activity from the application. To create a precise moment-in-time snapshot of a cluster, stop

all application write activity to the database, capture a backup, and allow only write operations to the database after the backup is complete.

However, you can capture a backup of a cluster that **approximates** a point-in-time backup by capturing a backup from a secondary member of the replica sets that provide the shards in the cluster at roughly the same moment. If you decide to use an approximate-point-in-time backup method, ensure that your application can operate using a copy of the data that does not reflect a single moment in time.

The following documents describe sharded cluster related backup procedures:

- *Backup a Small Sharded Cluster with mongodump* (page 331)
- *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 332)
- *Create Backup of a Sharded Cluster with Database Dumps* (page 333)
- *Schedule Backup Window for Sharded Clusters* (page 336)
- *Restore a Single Shard* (page 334)
- *Restore Sharded Clusters* (page 335)

Replica Set Backup Considerations

In most cases, backing up data stored in a *replica set* is similar to backing up data stored in a single instance. It is possible to lock a single *secondary* database and then create a backup from that instance. When you unlock the database, the secondary will catch up with the *primary*. You may also choose to deploy a dedicated *hidden member* for backup purposes.

If you have a *sharded cluster* where each *shard* is itself a replica set, you can use this method to create a backup of the entire cluster without disrupting the operation of the node. In these situations you should still turn off the balancer when you create backups.

For any cluster, using a non-primary node to create backups is particularly advantageous in that the backup operation does not affect the performance of the primary. Replication itself provides some measure of redundancy. Nevertheless, keeping point-in time backups of your cluster to provide for disaster recovery and as an additional layer of protection is crucial.

3.9 UNIX `ulimit` Settings

Most UNIX-like operating systems, including Linux and OS X, provide ways to limit and control the usage of system resources such as threads, files, and network connections on a per-process and per-user basis. These limits prevent single users from using too many system resources. Sometimes, these limits have low default values that can cause a number of issues in the course of normal MongoDB operation.

3.9.1 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, like other processes:

- track each incoming connection with a file descriptor *and* a thread.
- track each internal thread or *pthread* as a system process.

mongod

- 1 file descriptor for each data file in use by the `mongod` instance.
- 1 file descriptor for each journal file used by the `mongod` instance when `journal` 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*, replication, and replica set health checks, which may require a small number of additional resources.

mongos

In addition to the threads and file descriptors for client connections, `mongos` must maintain connects to all config servers and all shards, which includes all members of all replica sets.

For `mongos`, consider the following behaviors:

- `mongos` instances maintain a connection pool to each shard so that the `mongos` can reuse connections and quickly fulfill requests without needing to create new connections.
- You can limit the number of incoming connections using the `maxConns` run-time option:

  ```
  :option:`--maxConns <mongos --maxConns>`
  ```

  By restricting the number of incoming connections you can prevent a cascade effect where the `mongos` creates too many connections on the `mongod` instances.

  Note: You cannot set `maxConns` to a value higher than *20000*.

3.9.2 Review and Set Resource Limits

ulimit

You can use the `ulimit` command at the system prompt to check system limits, as in the following example:

```
$ ulimit -a
-t: cpu time (seconds)         unlimited
-f: file size (blocks)         unlimited
-d: data seg size (kbytes)     unlimited
-s: stack size (kbytes)        8192
-c: core file size (blocks)    0
-m: resident set size (kbytes) unlimited
-u: processes                  192276
-n: file descriptors           21000
-l: locked-in-memory size (kb) 40000
-v: address space (kb)         unlimited
-x: file locks                 unlimited
-i: pending signals            192276
-q: bytes in POSIX msg queues  819200
-e: max nice                   30
-r: max rt priority            65
-N 15:                         unlimited
```

`ulimit` refers to the per-*user* limitations for various resources. Therefore, if your `mongod` instance executes as a user that is also running multiple processes, or multiple `mongod` processes, you might see contention for these

resources. Also, be aware that the `processes` value (i.e. `-u`) refers to the combined number of distinct processes and sub-process threads.

You can change `ulimit` settings by issuing a command in the following form:

```
ulimit -n <value>
```

For many distributions of Linux you can change values by substituting the `-n` option for any possible value in the output of `ulimit -a`. On OS X, use the `launchctl limit` command. See your operating system documentation for the precise procedure for changing system limits on running systems.

Note: After changing the `ulimit` settings, you *must* restart the process to take advantage of the modified settings. You can use the `/proc` file system to see the current limitations on a running process.

Depending on your system's configuration, and default settings, any change to system limits made using `ulimit` may revert following system a system restart. Check your distribution and operating system documentation for more information.

`/proc` File System

Note: This section applies only to Linux operating systems.

The `/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:

```
return-limits mongod
return-limits mongos
return-limits mongod mongos
```

The output of the first command may resemble the following:

```
[mongod #6809 -- limits]
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
```

```
Max data size             unlimited           unlimited           bytes
Max stack size            8720000             unlimited           bytes
Max core file size        0                   unlimited           bytes
Max resident set          unlimited           unlimited           bytes
Max processes             192276              192276              processes
Max open files            1024                4096                files
Max locked memory         40960000            40960000            bytes
Max address space         unlimited           unlimited           bytes
Max file locks            unlimited           unlimited           locks
Max pending signals       192276              192276              signals
Max msgqueue size         819200              819200              bytes
Max nice priority         30                  30
Max realtime priority     65                  65
Max realtime timeout      unlimited           unlimited           us
```

3.9.3 Recommended Settings

Every deployment may have unique requirements and settings; however, the following thresholds and settings are particularly important for `mongod` and `mongos` deployments:

- `-f` (file size): `unlimited`
- `-t` (cpu time): `unlimited`
- `-v` (virtual memory): `unlimited`
- `-n` (open files): `64000`
- `-m` (memory size): `unlimited` [32]
- `-u` (processes/threads): `32000`

Always remember to restart your `mongod` and `mongos` instances after changing the `ulimit` settings to make sure that the settings change takes effect.

3.10 Production Notes

3.10.1 Overview

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

3.10.2 Backups

To make backups of your MongoDB database, please refer to *Backup Strategies for MongoDB Systems* (page 120).

3.10.3 Networking

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

See documents in the *Security* (page 133) section for additional information, specifically:

[32] If you limit the resident memory size on a system running MongoDB the operating system will refuse to honor additional allocation requests.

- *Interfaces and Port Numbers* (page 134)
- *Firewalls* (page 135)
- *Configure Linux iptables Firewall for MongoDB* (page 139)
- *Configure Windows netsh Firewall for MongoDB* (page 143)

For Windows users, consider the Windows Server Technet Article on TCP Configuration[33] when deploying MongoDB on Windows.

3.10.4 MongoDB on Linux

If you use the Linux kernel, the MongoDB user community has recommended Linux kernel 2.6.36 or later for running MongoDB in production.

Because MongoDB preallocates its database files before using them and because MongoDB uses very large files on average, you should use the Ext4 and XFS file systems if using the Linux kernel:

- If you use the Ext4 file system, use at least version 2.6.23 of the Linux Kernel.
- If you use the XFS file system, use at least version 2.6.25 of the Linux Kernel.

For MongoDB on Linux use the following recommended configurations:

- Turn off `atime` for the storage volume with the *database files*.
- Set the file descriptor limit and the user process limit above 20,000, according to the suggestions in *UNIX ulimit Settings* (page 122). A low ulimit will affect MongoDB when under heavy use and will produce weird errors.
- Do not use `hugepages` virtual memory pages, MongoDB performs better with normal virtual memory pages.
- Disable NUMA in your BIOS. If that is not possible see *NUMA* (page 128).
- Ensure that readahead settings for the block devices that store the database files are acceptable. See the *Readahead* (page 126) section
- Use NTP to synchronize time among your hosts. This is especially important in sharded clusters.

3.10.5 Readahead

For random access use patterns set readahead values low, for example setting readahead to a small value such as 32 (16KB) often works well.

3.10.6 MongoDB on Virtual Environments

The section describes considerations when running MongoDB in some of the more common virtual environments.

EC2

MongoDB is compatible with EC2 and requires no configuration changes specific to the environment.

[33] http://technet.microsoft.com/en-us/library/dd349797.aspx

VMWare

MongoDB is compatible with VMWare. Some in the MongoDB community have run into issues with VMWare's memory overcommit feature and suggest disabling the feature.

You can clone a virtual machine running MongoDB. You might use this to spin up a new virtual host that will be added as a member of a replica set. If journaling is enabled, the clone snapshot will be consistent. If not using journaling, stop `mongod`, clone, and then restart.

OpenVZ

The MongoDB community has encountered issues running MongoDB on OpenVZ.

3.10.7 Disk and Storage Systems

Swap

Configure swap space for your systems. Having swap can prevent issues with memory contention and can prevent the OOM Killer on Linux systems from killing `mongod`. Because of the way `mongod` maps memory files to memory, the operating system will never store MongoDB data in swap.

RAID

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

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

RAID-0 provides good write performance but provides limited availability, and reduced performance on read operations, particularly using Amazon's EBS volumes: as a result, avoid RAID-0 with MongoDB deployments.

Remote Filesystems

Some versions of NFS perform very poorly with MongoDB and NFS is not recommended for use with MongoDB. Performance problems arise when both the data files and the journal files are both hosted on NFS: you may experience better performance if you place the journal on local or `iscsi` volumes. If you must use NFS, add the following NFS options to your `/etc/fstab` file: `bg`, `nolock`, and `noatime`.

Many MongoDB deployments work successfully with Amazon's *Elastic Block Store* (EBS) volumes. There are certain intrinsic performance characteristics, with EBS volumes that users should consider.

3.10.8 Hardware Requirements and Limitations

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

When installing hardware for MongoDB, consider the following:

- As with all software, more RAM and a faster CPU clock speed are important to productivity.
- Because databases do not perform high amounts of computation, increasing the number cores helps but does not provide a high level of marginal return.

- MongoDB has good results and good price/performance with SATA SSD (Solid State Disk) and with PCI (Peripheral Component Interconnect).
- Commodity (SATA) spinning drives are often a good option as the speed increase for random I/O for more expensive drives is not that dramatic (only on the order of 2x), spending that money on SSDs or RAM may be more effective.

MongoDB on NUMA Hardware

Important: The discussion of NUMA in this section does not apply to deployments where `mongod` instances run on Windows.

MongoDB and NUMA, Non-Uniform Access Memory, do not work well together. When running MongoDB on NUMA hardware, disable NUMA for MongoDB and run with an interleave memory policy. NUMA can cause a number of operational problems with MongoDB, including slow performance for periods of time or high system processor usage.

Note: On Linux, MongoDB version 2.0 and greater checks these settings on start up and prints a warning if the system is NUMA-based.

To disable NUMA for MongoDB, use the `numactl` command and start `mongod` in the following manner:

```
numactl --interleave=all /usr/bin/local/mongod
```

Adjust the `proc` settings using the following command:

```
echo 0 > /proc/sys/vm/zone_reclaim_mode
```

To fully disable NUMA you must perform both operations. However, you can change `zone_reclaim_mode` without restarting mongod. For more information, see documentation on Proc/sys/vm[34].

See the The MySQL "swap insanity" problem and the effects of NUMA[35] post, which describes the effects of NUMA on databases. This blog post addresses the impact of NUMA for MySQL; however, the issues for MongoDB are similar. The post introduces NUMA its goals, and illustrates how these goals are not compatible with production databases.

3.10.9 Performance Monitoring

iostat

On Linux, use the iostat command to check if disk I/O is a bottleneck for your database. Specify a number of seconds when running iostat to avoid displaying stats covering the time since server boot.

For example:

```
iostat -xmt 1
```

Use the mount command to see what device your `data directory` resides on.

Key fields from `iostat`:

- `%util`: this is the most useful field for a quick check, it indicates what percent of the time the device/drive is in use.

[34] http://www.kernel.org/doc/Documentation/sysctl/vm.txt
[35] http://jcole.us/blog/archives/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/

- `avgrq-sz`: average request size. Smaller number for this value reflect more random IO operations.

bwm-ng

bwm-ng[36] 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.

3.10.10 Production Checklist

64-bit Builds for Production

Always use 64-bit Builds for Production. MongoDB uses memory mapped files. See the *32-bit limitations* (page 358) for more information.

32-bit builds exist to support use on development machines and also for other miscellaneous things such as replica set arbiters.

BSON Document Size Limit

There is a `BSON Document Size` limit – at the time of this writing 16MB per document. If you have large objects, use *GridFS* (page 58) instead.

Set Appropriate Write Concern for Write Operations

See *write concern* (page 38) for more information.

Dynamic Schema

Data in MongoDB has a *dynamic schema*. *Collections* do not enforce *document* structure. This facilitates iterative development and polymorphism. However, collections often hold documents with highly homogeneous structures. See *Data Modeling Considerations for MongoDB Applications* (page 43) for more information.

Some operational considerations include:

- the exact set of collections to be used
- the indexes to be used, which are created explicitly except for the `_id` index
- shard key declarations, which are explicit and quite important as it is hard to change shard keys later

One very simple rule-of-thumb is not to import data from a relational database unmodified: you will generally want to "roll up" certain data into richer documents that use some embedding of nested documents and arrays (and/or arrays of subdocuments).

Updates by Default Affect Only one Document

Set the `multi` parameter to `true` to `update` multiple documents that meet the query criteria. The `mongo` shell syntax is:

```
db.my_collection_name.update(my_query, my_update_expression, bool_upsert, bool_multi)
```

Set `bool_multi` to `true` when updating many documents. Otherwise only the first matched will update.

[36] http://www.gropp.org/?id=projects&sub=bwm-ng

Case Sensitive Strings

MongoDB strings are case sensitive. So a search for `"joe"` will not find `"Joe"`.

Consider:

- storing data in a normalized case format, or
- using regular expressions ending with `http://docs.mongodb.org/manual/i`
- and/or using `$toLower` or `$toUpper` in the *aggregation framework* (page 151)

Type Sensitive Fields

MongoDB data – which is JSON-style, specifically, BSON[37] format – have several data types.

Consider the following document which has a field `x` with the *string* value `"123"`:

```
{ x : "123" }
```

Then the following query which looks for a *number* value `123` will **not** return that document:

```
db.mycollection.find( { x : 123 } )
```

Locking

Older versions of MongoDB used a "global lock"; use MongoDB v2.2+ for better results. See the *Concurrency* (page 370) page for more information.

Packages

Be sure you have the latest stable release if you are using a package manager. You can see what is current on the Downloads page, even if you then choose to install via a package manager.

Use Odd Number of Replica Set Members

Replica sets (page 217) perform consensus elections. Use either an odd number of members (e.g., three) or else use an arbiter to get up to an odd number of votes.

Don't disable journaling

See *Journaling* (page 100) for more information.

Keep Replica Set Members Up-to-Date

This is important as MongoDB replica sets support automatic failover. Thus you want your secondaries to be up-to-date. You have a few options here:

1. Monitoring and alerts for any lagging can be done via various means. MMS shows a graph of replica set lag
2. Using *getLastError* (page 241) with `w:'majority'`, you will get a timeout or no return if a majority of the set is lagging. This is thus another way to guard against lag and get some reporting back of its occurrence.

[37] http://docs.mongodb.org/meta-driver/latest/legacy/bson/

3. Or, if you want to fail over manually, you can set your secondaries to `priority:0` in their configuration. Then manual action would be required for a failover. This is practical for a small cluster; for a large cluster you will want automation.

Additionally, see information on *replica set rollbacks* (page 219).

Additional Deployment Considerations

- Pick your shard keys carefully! There is no way to modify a shard key on a collection that is already sharded.
- You cannot shard an existing collection over 256 gigabytes. To shard large amounts of data, create a new empty sharded collection, and ingest the data from the source collection using an application level import operation.
- Unique indexes are not enforced across shards except for the shard key itself. See *Enforce Unique Keys for Sharded Collections* (page 338).
- Consider *pre-splitting* (page 298) a sharded collection before a massive bulk import. Usually this isn't necessary but on a bulk import of size it is helpful.
- Use *security/auth* (page 133) mode if you need it. By default `auth` is not enabled and `mongod` assumes a trusted environment.
- You do not have `fully generalized transactions`. Create rich documents and read the preceding link and consider the use case – often there is a good fit.
- Disable NUMA for best results. If you have NUMA enabled, `mongod` will print a warning when it starts.
- Avoid excessive prefetch/readahead on the filesystem. Check your prefetch settings. Note on linux the parameter is in *sectors*, not bytes. 32KBytes (a setting of 64 sectors) is pretty reasonable.
- Check *ulimit* (page 122) settings.
- Use SSD if available and economical. Spinning disks can work well but SSDs' capacity for random I/O operations work well with the update model of `mongod`. See *Remote Filesystems* (page 127) for more info.
- Ensure that clients keep reasonable pool sizes to avoid overloading the connection tracking capacity of a single `mongod` or `mongos` instance.

See also:

- *Replica Set Operation and Management* (page 223)
- *Replica Set Architectures and Deployment Patterns* (page 238)
- *Sharded Cluster Administration* (page 298)
- *Sharded Cluster Architectures* (page 302)
- *Tag Aware Sharding* (page 336)
- *Indexing Overview* (page 185)
- *Indexing Operations* (page 194)

Additionally, Consider the `http://docs.mongodb.org/manual/tutorial` page that contains a full index of all tutorials available in the MongoDB manual. These documents provide pragmatic instructions for common operational practices and administrative tasks.

CHAPTER 4

Security

The documents outline basic security practices and risk management strategies. Additionally, this section includes *tutorials* that outline basic network filter and firewall rules to configure trusted environments for MongoDB.

4.1 Strategies and Practices

4.1.1 Security Practices and Management

As with all software running in a networked environment, administrators of MongoDB must consider security and risk exposures for a MongoDB deployment. There are no magic solutions for risk mitigation, and maintaining a secure MongoDB deployment is an ongoing process. This document takes a *Defense in Depth* approach to securing MongoDB deployments, and addresses a number of different methods for managing risk and reducing risk exposure

The intent of *Defense In Depth* approaches are to ensure there are no exploitable points of failure in your deployment that could allow an intruder or un-trusted party to access the data stored in the MongoDB database. The easiest and most effective way to reduce the risk of exploitation is to run MongoDB in a trusted environment, limit access, follow a system of least privilege, and follow best development and deployment practices. See the *Strategies for Reducing Risk* (page 133) section for more information.

Strategies for Reducing Risk

The most effective way to reduce risk for MongoDB deployments is to run your entire MongoDB deployment, including all MongoDB components (i.e. `mongod`, `mongos` and application instances) in a *trusted environment*. Trusted environments use the following strategies to control access:

- network filter (e.g. firewall) rules that block all connections from unknown systems to MongoDB components.
- bind `mongod` and `mongos` instances to specific IP addresses to limit accessibility.
- limit MongoDB programs to non-public local networks, and virtual private networks.

You may further reduce risk by:

- requiring authentication for access to MongoDB instances.
- requiring strong, complex, single purpose authentication credentials. This should be part of your internal security policy but is not currently configurable in MongoDB.
- deploying a model of least privilege, where all users have *only* the amount of access they need to accomplish required tasks, and no more.

- following the best application development and deployment practices, which includes: validating all inputs, managing sessions, and application-level access control.

Continue reading this document for more information on specific strategies and configurations to help reduce the risk exposure of your application.

Vulnerability Notification

MongoDB, Inc. takes the security of MongoDB and associated products very seriously. If you discover a vulnerability in MongoDB, or would like to know more about our vulnerability reporting and response process, see the *Vulnerability Notification* (page 138) document.

Networking Risk Exposure

Interfaces and Port Numbers

The following list includes all default ports used by MongoDB:

By default, listens for connections on the following ports:

27017 This is the default port `mongod` and `mongos` instances. You can change this port with `port` or `--port`.

27018 This is the default port when running with `--shardsvr` runtime operation or `shardsvr` setting.

27019 This is the default port when running with `--configsvr` runtime operation or `configsvr` setting.

28017 This is the default port for the web status page. This is always accessible at a port that is `1000` greater than the port determined by `port`.

By default MongoDB programs (i.e. `mongos` and `mongod`) will bind to all available network interfaces (i.e. IP addresses) on a system. The next section outlines various runtime options that allow you to limit access to MongoDB programs.

Network Interface Limitation

You can limit the network exposure with the following configuration options:

- the `nohttpinterface` setting for `mongod` and `mongos` instances.

  Disables the "home" status page, which would run on port `28017` by default. The status interface is read-only by default. You may also specify this option on the command line as `mongod --nohttpinterface` or `mongos --nohttpinterface`. Authentication does not control or affect access to this interface.

  Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

- the `port` setting for `mongod` and `mongos` instances.

  Changes the main port on which the `mongod` or `mongos` instance listens for connections. Changing the port does not meaningfully reduce risk or limit exposure.

  You may also specify this option on the command line as `mongod --port` or `mongos --port`.

  Whatever port you attach `mongod` and `mongos` instances to, you should only allow trusted clients to connect to this port.

- the `rest` setting for `mongod`.

  Enables a fully interactive administrative *REST* interface, which is *disabled by default.* The status interface, which *is* enabled by default, is read-only. This configuration makes that interface fully interactive. The REST interface does not support any authentication and you should always restrict access to this interface to only allow trusted clients to connect to this port.

  You may also enable this interface on the command line as *`mongod --rest`*.

  Important: Disable this option for production deployments. If *do* you leave this interface enabled, you should only allow trusted clients to access this port.

- the `bind_ip` setting for `mongod` and `mongos` instances.

  Limits the network interfaces on which MongoDB programs will listen for incoming connections. You can also specify a number of interfaces by passing `bind_ip` a comma separated list of IP addresses. You can use the *`mongod --bind_ip`* and *`mongos --bind_ip`* option on the command line at run time to limit the network accessibility of a MongoDB program.

  Important: Make sure that your `mongod` and `mongos` instances are only accessible on trusted networks. If your system has more than one network interface, bind MongoDB programs to the private or internal network interface.

Firewalls

Firewalls allow administrators to filter and control access to a system by providing granular control over what network communications. For administrators of MongoDB, the following capabilities are important:

- limiting incoming traffic on a specific port to specific systems.
- limiting incoming traffic from untrusted hosts.

On Linux systems, the `iptables` interface provides access to the underlying `netfilter` firewall. On Windows systems `netsh` command line interface provides access to the underlying Windows Firewall. For additional information about firewall configuration consider the following documents:

- *Configure Linux iptables Firewall for MongoDB* (page 139)
- *Configure Windows netsh Firewall for MongoDB* (page 143)

For best results and to minimize overall exposure, ensure that *only* traffic from trusted sources can reach `mongod` and `mongos` instances and that the `mongod` and `mongos` instances can only connect to trusted outputs.

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2[1] page, which addresses Amazon's Security Groups and other EC2-specific security features.

Virtual Private Networks

Virtual private networks, or VPNs, make it possible to link two networks over an encrypted and limited-access trusted network. Typically MongoDB users who use VPNs use SSL rather than IPSEC VPNs for performance issues.

Depending on configuration and implementation VPNs provide for certificate validation and a choice of encryption protocols, which requires a rigorous level of authentication and identification of all clients. Furthermore, because

[1] http://docs.mongodb.org/ecosystem/platforms/amazon-ec2

VPNs provide a secure tunnel, using a VPN connection to control access to your MongoDB instance, you can prevent tampering and "man-in-the-middle" attacks.

Operations

Always run the `mongod` or `mongos` process as a *unique* user with the minimum required permissions and access. Never run a MongoDB program as a `root` or administrative users. The system users that run the MongoDB processes should have robust authentication credentials that prevent unauthorized or casual access.

To further limit the environment, you can run the `mongod` or `mongos` process in a `chroot` environment. Both user-based access restrictions and `chroot` configuration follow recommended conventions for administering all daemon processes on Unix-like systems.

You can disable anonymous access to the database by enabling authentication using the `auth` as detailed in the *Authentication* (page 136) section.

Authentication

MongoDB provides basic support for authentication with the `auth` setting. For multi-instance deployments (i.e. *replica sets*, and *sharded clusters*) use the `keyFile` setting, which implies `auth`, and allows intra-deployment authentication and operation. Be aware of the following behaviors of MongoDB's authentication system:

- Authentication is **disabled** by default.
- MongoDB provisions access on a per-database level. Users either have *read only* access to a database or *normal* access to a database that permits full read and write access to the database. *Normal* access conveys the ability to add additional users to the database.
- The `system.users` collection in each database stores all credentials. You can query the authorized users with the following operation:

  ```
  db.system.users.find()
  ```

- The `admin` database is unique. Users with *normal* access to the `admin` database have read and write access to all databases. Users with *read only* access to the `admin` database have read only access to all databases, with the exception of the `system.users` collection, which is protected to prevent privilege escalation attacks.

  Additionally the `admin` database exposes several commands and functionality, such as `listDatabases`.
- Once authenticated a *normal* user has full read and write access to a database.
- If you have authenticated to a database as a normal, read and write, user; authenticating as a read-only user on the same database will invalidate the earlier authentication, leaving the current connection with read only access.
- If you have authenticated to the `admin` database as normal, read and write, user; logging into a *different* database as a read only user will *not* invalidate the authentication to the `admin` database. In this situation, this client will be able to read and write data to this second database.
- When setting up authentication for the first time you must either:
  1. add at least one user to the `admin` database before starting the `mongod` instance with `auth`.
  2. add the first user to the `admin` database when connected to the `mongod` instance from a `localhost` connection. [2]

[2] Because of SERVER-6591 (https://jira.mongodb.org/browse/SERVER-6591), you cannot add the first user to a sharded cluster using the `localhost` connection in 2.2. If you are running a 2.2 sharded cluster, and want to enable authentication, you must deploy the cluster and add the first user to the `admin` database before restarting the cluster to run with `keyFile`.

New in version 2.0: Support for authentication with sharded clusters. Before 2.0 sharded clusters *had* to run with trusted applications and a trusted networking configuration.

Consider the *Control Access to MongoDB Instances with Authentication* (page 146) document which outlines procedures for configuring and maintaining users and access with MongoDB's authentication system.

Interfaces

Simply limiting access to a `mongod` is not sufficient for totally controlling risk exposure. Consider the recommendations in the following section, for limiting exposure other interface-related risks.

JavaScript and the Security of the `mongo` Shell

Be aware of the following capabilities and behaviors of the `mongo` shell:

- `mongo` will evaluate a `.js` file passed to the *`mongo --eval`* option. The `mongo` shell does not validate the input of JavaScript input to *`--eval`*.
- `mongo` will evaluate a `.mongorc.js` file before starting. You can disable this behavior by passing the *`mongo --norc`* option.

  On Linux and Unix systems, `mongo` reads the `.mongorc.js` file from *`$HOME/`*`.mongorc.js` (i.e. `~/.mongorc.js`), and Windows `mongo.exe` reads the `.mongorc.js` file from *`%HOME%`*`.mongorc.js` or *`%HOMEDRIVE%%HOMEPATH%`*`.mongorc.js`.

HTTP Status Interface

The HTTP status interface provides a web-based interface that includes a variety of operational data, logs, and status reports regarding the `mongod` or `mongos` instance. The HTTP interface is always available on the port numbered `1000` greater than the primary `mongod` port. By default this is `28017`, but is indirectly set using the `port` option which allows you to configure the primary `mongod` port.

Without the `rest` setting, this interface is entirely read-only, and limited in scope; nevertheless, this interface may represent an exposure. To disable the HTTP interface, set the `nohttpinterface` run time option or the *`--nohttpinterface`* command line option.

REST API

The REST API to MongoDB provides additional information and write access on top of the HTTP Status interface. The REST interface is *disabled* by default, and is not recommended for production use.

While the REST API does not provide any support for insert, update, or remove operations, it does provide administrative access, and its accessibility represents a vulnerability in a secure environment.

If you must use the REST API, please control and limit access to the REST API. The REST API does not include any support for authentication, even if when running with `auth` enabled.

See the following documents for instructions on restricting access to the REST API interface:

- *Configure Linux iptables Firewall for MongoDB* (page 139)
- *Configure Windows netsh Firewall for MongoDB* (page 143)

Data Encryption

To support audit requirements, you may need to encrypt data stored in MongoDB. For best results you can encrypt this data in the application layer, by encrypting the content of fields that hold secure data.

4.1.2 Vulnerability Notification

MongoDB[3] values the privacy and security of all users of MongoDB, and every effort is made to ensure that MongoDB and related tools minimize risk exposure and increase the security and integrity of data and environments using MongoDB.

Notification

If you believe you have discovered a vulnerability in MongoDB or have experienced a security incident related to MongoDB, please report the issue so it can be avoided in future. All vulnerability reports should contain as much information as possible so that the issue can be resolved quickly. 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.

All vulnerability notifications are responded to within 48 hours.

Jira

jira.mongodb.org[4] is the perferred method of communication regarding MongoDB.

Submit a ticket in the Core Server Security[5]" project, at: <https://jira.mongodb.org/browse/SECURITY/>. The ticket number will become reference identification for the issue for the lifetime of the issue, and you can use this identifier for tracking purposes.

MongoDB, Inc. will respond to any vulnerability notification received in a Jira case posted to the SECURITY[6] project.

Email

While Jira is preferred, you may also report vulnerabilities via email to <security@mongodb.com[7]>.

You may encrypt email using MongoDB's' public key[8], to ensure the privacy of any sensitive information in your vulnerability report.

MongoDB, Inc. will respond to any vulnerability notification received via email with email which will contain a reference number (i.e. a ticket from the SECURITY[9] project,) Jira case posted to the SECURITY[10] project.

[3] http://www.mongodb.com/
[4] https://jira.mongodb.org
[5] https://jira.mongodb.org/browse/SECURITY
[6] https://jira.mongodb.org/browse/SECURITY
[7] security@mongodb.com
[8] http://docs.mongodb.org/10gen-gpg-key.asc
[9] https://jira.mongodb.org/browse/SECURITY
[10] https://jira.mongodb.org/browse/SECURITY

Evaluation

MongoDB, Inc. validates all submitted vulnerabilities and uses Jira to track all communications regarding the vulnerability, including requests for clarification and for additional information. If needed, MongoDB representatives can 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, respond to the notification, and to notify key users, customers, and partners if needed.

The amount of time required to validate a reported vulnerability depends on the complexity and severity of the issue. 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 of the vulnerability.

After validating the issue, MongoDB, Inc. will coordinate public disclosure of the issue with the reporter in a mutually agreed timeframe and format. If required or requested, the reporter of a vulnerability will receive credit in the published security bulletin.

4.2 Tutorials

4.2.1 Configure Linux `iptables` Firewall for MongoDB

On contemporary Linux systems, the `iptables` program provides methods for managing the Linux Kernel's `netfilter` or network packet filtering capabilities. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic firewall configurations for `iptables` firewalls on Linux. Use these approaches as a starting point for your larger networking organization. For a detailed over view of security practices and risk management for MongoDB, see *Security Practices and Management* (page 133).

See also:

For MongoDB deployments on Amazon's web services, see the Amazon EC2[11] 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 only the following two chains:

`INPUT` Controls all incoming traffic.

`OUTPUT` Controls all outgoing traffic.

Given the *default ports* (page 134) 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

[11] http://docs.mongodb.org/ecosystem/platforms/amazon-ec2

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 142).

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 -
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27017 -m state --state ESTABLISHED -j ACCEPT
```

The first rule allows all incoming traffic from `<ip-address>` on port `27017`, which allows the application server to connect to the `mongod` instance. The second rule, allows outgoing traffic from the `mongod` to reach the application server.

Optional

If you have only one application server, you can replace `<ip-address>` with either the IP address itself, such as: `198.51.100.55`. You can also express this using CIDR notation as `198.51.100.55/32`. If you want to permit a larger block of possible IP addresses you can allow traffic from a `/24` using one of the following specifications for the `<ip-address>`, as follows:

```
10.10.10.10/24
10.10.10.10/255.255.255.0
```

Traffic to and from `mongos` Instances

`mongos` instances provide query routing for *sharded clusters*. Clients connect to `mongos` instances, which behave from the client's perspective as `mongod` instances. In turn, the `mongos` connects to all `mongod` instances that are components of the sharded cluster.

Use the same `iptables` command to allow traffic to and from these instances as you would from the `mongod` instances that are members of the replica set. Take the configuration outlined in the *Traffic to and from mongod Instances* (page 140) 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. [12] 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.

[12] You can also run a config server by setting the `configsvr` option in a configuration file.

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -
iptables -A OUTPUT -d <ip-address> -p tcp --source-port 27019 -m state --state ESTABLISHED -j ACCEPT
```

Replace `<ip-address>` with the address or address space of *all* the `mongod` that provide config servers.

Additionally, config servers need to allow incoming connections from all of the `mongos` instances in the cluster *and* all `mongod` instances in the cluster. Add rules that resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27019 -m state --state NEW,ESTABLISHED -
```

Replace `<ip-address>` with the address of the `mongos` instances and the shard `mongod` instances.

Traffic to and from a MongoDB Shard Server

For shard servers, running as *`mongod --shardsvr`* [13] Because the default port number when running with `shardsvr` is `27018`, you must configure the following `iptables` rules to allow traffic to and from each shard:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 27018 -m state --state NEW,ESTABLISHED -
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. [14]

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,ESTABLISH
   ```

   Replace `<ip-address>` with the address of the instance that needs access to the HTTP or REST interface. For *all* deployments, you should restrict access to this port to *only* the monitoring instance.

   Optional

   For shard server `mongod` instances running with `shardsvr`, the rule would resemble the following:

   ```
   iptables -A INPUT -s <ip-address> -p tcp --destination-port 28018 -m state --state NEW,ESTABLISH
   ```

[13] You can also specify the shard server option using the `shardsvr` setting in the configuration file. Shard members are also often conventional replica sets using the default port.

[14] 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 `configsvr`, the rule would resemble the following:

```
iptables -A INPUT -s <ip-address> -p tcp --destination-port 28019 -m state --state NEW,ESTABLISH
```

Change Default Policy to `DROP`

The default policy for `iptables` chains is to allow all traffic. After completing all `iptables` configuration changes, you *must* change the default policy to `DROP` so that all traffic that isn't explicitly allowed as above will not be able to reach components of the MongoDB deployment. Issue the following commands to change this policy:

```
iptables -P INPUT DROP

iptables -P OUTPUT DROP
```

Manage and Maintain `iptables` Configuration

This section contains a number of basic operations for managing and using `iptables`. There are various front end tools that automate some aspects of `iptables` configuration, but at the core all `iptables` front ends provide the same basic functionality:

Make all `iptables` Rules Persistent

By default all `iptables` rules are only stored in memory. When your system restarts, your firewall rules will revert to their defaults. When you have tested a rule set and have guaranteed that it effectively controls traffic you can use the following operations to you should make the rule set persistent.

On Red Hat Enterprise Linux, Fedora Linux, and related distributions you can issue the following command:

```
service iptables save
```

On Debian, Ubuntu, and related distributions, you can use the following command to dump the `iptables` rules to the `/etc/iptables.conf` file:

```
iptables-save > /etc/iptables.conf
```

Run the following operation to restore the network rules:

```
iptables-restore < /etc/iptables.conf
```

Place this command in your `rc.local` file, or in the `/etc/network/if-up.d/iptables` file with other similar operations.q

List all `iptables` Rules

To list all of currently applied `iptables` rules, use the following operation at the system shell.

```
iptables --L
```

Flush all `iptables` Rules

If you make a configuration mistake when entering `iptables` rules or simply need to revert to the default rule set, you can use the following operation at the system shell to flush all rules:

```
iptables --F
```

If you've already made your `iptables` rules persistent, you will need to repeat the appropriate procedure in the *Make all iptables Rules Persistent* (page 142) section.

4.2.2 Configure Windows `netsh` Firewall for MongoDB

On Windows Server systems, the `netsh` program provides methods for managing the *Windows Firewall*. These firewall rules make it possible for administrators to control what hosts can connect to the system, and limit risk exposure by limiting the hosts that can connect to a system.

This document outlines basic *Windows Firewall* configurations. Use these approaches as a starting point for your larger networking organization. For a detailed over view of security practices and risk management for MongoDB, see *Security Practices and Management* (page 133).

See also:

Windows Firewall[15] 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 134) 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.

[15] http://technet.microsoft.com/en-us/network/bb545423.aspx

Traffic to and from `mongod.exe` Instances

This pattern is applicable to all `mongod.exe` instances running as standalone instances or as part of a *replica set*. The goal of this pattern is to explicitly allow traffic to the `mongod.exe` instance from the application server.

```
netsh advfirewall firewall add rule name="Open mongod port 27017" dir=in action=allow protocol=TCP lo
```

This rule allows all incoming traffic to port `27017`, which allows the application server to connect to the `mongod.exe` instance.

Windows Firewall also allows enabling network access for an entire application rather than to a specific port, as in the following example:

```
netsh advfirewall firewall add rule name="Allowing mongod" dir=in action=allow program=" C:\mongodb\b
```

You can allow all access for a `mongos.exe` server, with the following invocation:

```
netsh advfirewall firewall add rule name="Allowing mongos" dir=in action=allow program=" C:\mongodb\b
```

Traffic to and from `mongos.exe` Instances

`mongos.exe` instances provide query routing for *sharded clusters*. Clients connect to `mongos.exe` instances, which behave from the client's perspective as `mongod.exe` instances. In turn, the `mongos.exe` connects to all `mongod.exe` instances that are components of the sharded cluster.

Use the same *Windows Firewall* command to allow traffic to and from these instances as you would from the `mongod.exe` instances that are members of the replica set.

```
netsh advfirewall firewall add rule name="Open mongod shard port 27018" dir=in action=allow protocol=
```

Traffic to and from a MongoDB Config Server

Configuration servers, host the *config database* that stores metadata for sharded clusters. Each production cluster has three configuration servers, initiated using the *`mongod --configsvr`* option. [16] Configuration servers listen for connections on port `27019`. As a result, add the following *Windows Firewall* rules to the config server to allow incoming and outgoing connection on port `27019`, for connection to the other config servers.

```
netsh advfirewall firewall add rule name="Open mongod config svr port 27019" dir=in action=allow prot
```

Additionally, config servers need to allow incoming connections from all of the `mongos.exe` instances in the cluster *and* all `mongod.exe` instances in the cluster. Add rules that resemble the following:

```
netsh advfirewall firewall add rule name="Open mongod config svr inbound" dir=in action=allow protoco
```

Replace `<ip-address>` with the addresses of the `mongos.exe` instances and the shard `mongod.exe` instances.

Traffic to and from a MongoDB Shard Server

For shard servers, running as *`mongod --shardsvr`* [17] Because the default port number when running with `shardsvr` is `27018`, you must configure the following *Windows Firewall* rules to allow traffic to and from each shard:

[16] You can also run a config server by setting the `configsvr` option in a configuration file.

[17] You can also specify the shard server option using the `shardsvr` setting in the configuration file. Shard members are also often conventional replica sets using the default port.

```
netsh advfirewall firewall add rule name="Open mongod shardsvr inbound" dir=in action=allow protocol=
netsh advfirewall firewall add rule name="Open mongod shardsvr outbound" dir=out action=allow protoco
```

Replace the `<ip-address>` specification with the IP address of all `mongod.exe` instances. This allows you to permit incoming and outgoing traffic between all shards including constituent replica set members to:

- all `mongod.exe` instances in the shard's replica sets.
- all `mongod.exe` instances in other shards. [18]

Furthermore, shards need to be able make outgoing connections to:

- all `mongos.exe` instances.
- all `mongod.exe` instances in the config servers.

Create a rule that resembles the following, and replace the `<ip-address>` with the address of the config servers and the `mongos.exe` instances:

```
netsh advfirewall firewall add rule name="Open mongod config svr outbound" dir=out action=allow proto
```

Provide Access For Monitoring Systems

1. The `mongostat` diagnostic tool, when running with the *`--discover`* needs to be able to reach all components of a cluster, including the config servers, the shard servers, and the `mongos.exe` instances.
2. If your monitoring system needs access the HTTP interface, insert the following rule to the chain:

   ```
   netsh advfirewall firewall add rule name="Open mongod HTTP monitoring inbound" dir=in action=all
   ```

   Replace `<ip-address>` with the address of the instance that needs access to the HTTP or REST interface. For *all* deployments, you should restrict access to this port to *only* the monitoring instance.

   Optional

   For shard server `mongod.exe` instances running with `shardsvr`, the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongos HTTP monitoring inbound" dir=in action=all
   ```

   For config server `mongod.exe` instances running with `configsvr`, the rule would resemble the following:

   ```
   netsh advfirewall firewall add rule name="Open mongod configsvr HTTP monitoring inbound" dir=in
   ```

Manage and Maintain *Windows Firewall* Configurations

This section contains a number of basic operations for managing and using `netsh`. While you can use the GUI front ends to manage the *Windows Firewall*, all core functionality is accessible is accessible from `netsh`.

Delete all *Windows Firewall* Rules

To delete the firewall rule allowing `mongod.exe` traffic:

```
netsh advfirewall firewall delete rule name="Open mongod port 27017" protocol=tcp localport=27017
```

```
netsh advfirewall firewall delete rule name="Open mongod shard port 27018" protocol=tcp localport=270
```

[18] All shards in a cluster need to be able to communicate with all other shards to facilitate *chunk* and balancing operations.

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"
```

4.2.3 Control Access to MongoDB Instances with Authentication

MongoDB provides a basic authentication system, that you can enable with the `auth` and `keyFile` configuration settings. [19] See the *authentication* (page 136) section of the *Security Practices and Management* (page 133) document.

This document contains an overview of all operations related to authentication and managing a MongoDB deployment with authentication.

See

The *Security Considerations* (page 96) section of the *Run-time Database Configuration* (page 95) document for more information on configuring authentication.

Add Users

When setting up authentication for the first time you must either:

1. add at least one user to the `admin` database before starting the `mongod` instance with `auth`.
2. add the first user to the `admin` database when connected to the `mongod` instance from a `localhost` connection. [20]

Begin by setting up the first administrative user for the `mongod` instance.

[19] Use the *`--auth --keyFile`* options on the command line.

[20] Because of SERVER-6591 (https://jira.mongodb.org/browse/SERVER-6591), you cannot add the first user to a sharded cluster using the `localhost` connection in 2.2. If you are running a 2.2 sharded cluster, and want to enable authentication, you must deploy the cluster and add the first user to the `admin` database before restarting the cluster to run with `keyFile`.

Add an Administrative User

About administrative users

Administrative users are those users that have "normal" or read and write access to the `admin` database.

If this is the first administrative user, [21] connect to the `mongod` on the `localhost` interface using the `mongo` shell. Then, issue the following command sequence to switch to the `admin` database context and add the administrative user:

```
use admin
db.addUser("<username>", "<password>")
```

Replace `<username>` and `<password>` with the credentials for this administrative user.

Add a Normal User to a Database

To add a user with read and write access to a specific database, in this example the `records` database, connect to the `mongod` instance using the `mongo` shell, and issue the following sequence of operations:

```
use records
db.addUser("<username>", "<password>")
```

Replace `<username>` and `<password>` with the credentials for this user.

Add a Read Only User to a Database

To add a user with read only access to a specific database, in this example the `records` database, connect to the `mongod` instance using the `mongo` shell, and issue the following sequence of operations:

```
use records
db.addUser("<username>", "<password>", true)
```

Replace `<username>` and `<password>` with the credentials for this user.

Administrative Access in MongoDB

Although administrative accounts have access to all databases, these users must authenticate against the `admin` database before changing contexts to a second database, as in the following example:

Example

Given the `superAdmin` user with the password `Password123`, and access to the `admin` database.

The following operation in the `mongo` shell will succeed:

```
use admin
db.auth("superAdmin", "Password123")
```

However, the following operation will fail:

```
use test
db.auth("superAdmin", "Password123")
```

[21] You can also use this procedure if authentication is *not* enabled so that your databases has an administrative user when you enable `auth`.

Note: If you have authenticated to the `admin` database as normal, read and write, user; logging into a *different* database as a read only user will *not* invalidate the authentication to the `admin` database. In this situation, this client will be able to read and write data to this second database.

Authentication on Localhost

The behavior of `mongod` running with `auth`, when connecting from a client over the localhost interface (i.e. a client running on the same system as the `mongod`,) varies slightly between before and after version 2.2.

In general if there are no users for the `admin` database, you may connect via the localhost interface. For sharded clusters running version 2.2, if `mongod` is running with `auth` then all users connecting over the localhost interface must authenticate, even if there aren't any users in the `admin` database.

Password Hashing Insecurity

In version 2.2 and earlier:

- the *normal* users of a database all have access to the `system.users` collection, which contains the user names and a hash of all user's passwords. [22]
- if a user has the same password in multiple databases, the hash will be the same on all database. A malicious user could exploit this to gain access on a second database use a different users' credentials.

As a result, always use unique username and password combinations on for each database.

Thanks to Will Urbanski, from Dell SecureWorks, for identifying this issue.

Configuration Considerations for Authentication

The following sections, outline practices for enabling and managing authentication with specific MongoDB deployments:

- *Security Considerations for Replica Sets* (page 232)
- *Sharded Cluster Security Considerations* (page 301)

Generate a Key File

The key file must be less than one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or "world" permissions on UNIX systems. Key file permissions are not checked on Windows systems.

Windows Systems

Use the following `openssl` command at the system shell to generate pseudo-random content for a key file for deployments with Windows components:

```
openssl rand -base64 741
```

[22] Read only users do not have access to the `system.users` database.

Linux and Unix Systems

Use the following `openssl` command at the system shell to generate pseudo-random content for a key file for systems that do not have Windows components (i.e. OS X, Unix, or Linux systems):

```
openssl rand -base64 753
```

Key File Properties

Be aware that MongoDB strips whitespace characters (e.g. `x0d`, `x09`, and `x20`,) for cross-platform convenience. As a result, the following operations produce identical keys:

```
echo -e "my secret key" > key1
echo -e "my secret key\n" > key2
echo -e "my    secret    key" > key3
echo -e "my\r\nsecret\r\nkey\r\n" > key4
```

CHAPTER 5

Aggregation

In version 2.2, MongoDB introduced the *aggregation framework* (page 151) that provides a powerful and flexible set of tools to use for many data aggregation tasks. If you're familiar with data aggregation in SQL, consider the `http://docs.mongodb.org/manual/reference/sql-aggregation-comparison` document as an introduction to some of the basic concepts in the aggregation framework. Consider the full documentation of the aggregation framework here:

5.1 Aggregation Framework

New in version 2.1.

5.1.1 Overview

The MongoDB aggregation framework provides a means to calculate aggregated values without having to use *map-reduce*. While map-reduce is powerful, it is often more difficult than necessary for many simple aggregation tasks, such as totaling or averaging field values.

If you're familiar with *SQL*, the aggregation framework provides similar functionality to `GROUP BY` and related SQL operators as well as simple forms of "self joins." Additionally, the aggregation framework provides projection capabilities to reshape the returned data. Using the projections in the aggregation framework, you can add computed fields, create new virtual sub-objects, and extract sub-fields into the top-level of results.

See also:

A presentation from MongoSV 2011: MongoDB's New Aggregation Framework[1].

Additionally, consider *Aggregation Framework Examples* (page 155) and *Aggregation Framework Reference* (page 163) for more documentation.

5.1.2 Framework Components

This section provides an introduction to the two concepts that underpin the aggregation framework: *pipelines* and *expressions*.

[1] http://www.mongodb.com/presentations/mongosv-2011/mongodbs-new-aggregation-framework

Pipelines

Conceptually, documents from a collection pass through an aggregation pipeline, which transforms these objects as they pass through. For those familiar with UNIX-like shells (e.g. bash,) the concept is analogous to the pipe (i.e. `|`) used to string text filters together.

In a shell environment the pipe redirects a stream of characters from the output of one process to the input of the next. The MongoDB aggregation pipeline streams MongoDB documents from one *pipeline operator* (page 164) to the next to process the documents. Pipeline operators can be repeated in the pipe.

All pipeline operators process a stream of documents and the pipeline behaves as if the operation scans a *collection* and passes all matching documents into the "top" of the pipeline. Each operator in the pipeline transforms each document as it passes through the pipeline.

Note: Pipeline operators need not produce one output document for every input document: operators may also generate new documents or filter out documents.

Warning: The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

See also:

The "*Aggregation Framework Reference* (page 163)" includes documentation of the following pipeline operators:

- `$project`
- `$match`
- `$limit`
- `$skip`
- `$unwind`
- `$group`
- `$sort`

Expressions

Expressions (page 170) produce output documents based on calculations performed on input documents. The aggregation framework defines expressions using a document format using prefixes.

Expressions are stateless and are only evaluated when seen by the aggregation process. All aggregation expressions can only operate on the current document in the pipeline, and cannot integrate data from other documents.

The *accumulator* expressions used in the `$group` operator maintain that state (e.g. totals, maximums, minimums, and related data) as documents progress through the *pipeline*.

See also:

Aggregation expressions (page 170) for additional examples of the expressions provided by the aggregation framework.

5.1.3 Use

Invocation

Invoke an *aggregation* operation with the `aggregate()` wrapper in the `mongo` shell or the `aggregate` *database command*. Always call `aggregate()` on a collection object that determines the input documents of the aggregation

pipeline. The arguments to the `aggregate()` method specify a sequence of *pipeline operators* (page 164), where each operator may have a number of operands.

First, consider a *collection* of documents named `articles` using the following format:

```
{
 title : "this is my title" ,
 author : "bob" ,
 posted : new Date () ,
 pageViews : 5 ,
 tags : [ "fun" , "good" , "fun" ] ,
 comments : [
             { author :"joe" , text : "this is cool" } ,
             { author :"sam" , text : "this is bad" }
 ],
 other : { foo : 5 }
}
```

The following example aggregation operation pivots data to create a set of author names grouped by tags applied to an article. Call the aggregation framework by issuing the following command:

```
db.articles.aggregate(
  { $project : {
     author : 1,
     tags : 1,
  } },
  { $unwind : "$tags" },
  { $group : {
     _id : { tags : "$tags" },
     authors : { $addToSet : "$author" }
  } }
);
```

The aggregation pipeline begins with the *collection* `articles` and selects the `author` and `tags` fields using the `$project` aggregation operator. The `$unwind` operator produces one output document per tag. Finally, the `$group` operator pivots these fields.

Result

The aggregation operation in the previous section returns a *document* with two fields:

- `result` which holds an array of documents returned by the *pipeline*
- `ok` which holds the value `1`, indicating success, or another value if there was an error

As a document, the result is subject to the *BSON Document size* limit, which is currently 16 megabytes.

5.1.4 Optimizing Performance

Because you will always call `aggregate` on a *collection* object, which logically inserts the *entire* collection into the aggregation pipeline, you may want to optimize the operation by avoiding scanning the entire collection whenever possible.

Pipeline Operators and Indexes

Depending on the order in which they appear in the pipeline, aggregation operators can take advantage of indexes.

The following pipeline operators take advantage of an index when they occur at the beginning of the pipeline:

- `$match`
- `$sort`
- `$limit`
- `$skip`.

The above operators can also use an index when placed **before** the following aggregation operators:

- `$project`
- `$unwind`
- `$group`.

Early Filtering

If your aggregation operation requires only a subset of the data in a collection, use the `$match` operator to restrict which items go in to the top of the pipeline, as in a query. When placed early in a pipeline, these `$match` 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.

In future versions there may be an optimization phase in the pipeline that reorders the operations to increase performance without affecting the result. However, at this time place `$match` operators at the beginning of the pipeline when possible.

Memory for Cumulative Operators

Certain pipeline operators require access to the entire input set before they can produce any output. For example, `$sort` must receive all of the input from the preceding *pipeline* operator before it can produce its first output document. The current implementation of `$sort` does not go to disk in these cases: in order to sort the contents of the pipeline, the entire input must fit in memory.

`$group` has similar characteristics: Before any `$group` passes its output along the pipeline, it must operator, this frequently does not require as much memory as `$sort`, because it only needs to retain one record for each unique key in the grouping specification.

The current implementation of the aggregation framework logs a warning if a cumulative operator consumes 5% or more of the physical memory on the host. Cumulative operators produce an error if they consume 10% or more of the physical memory on the host.

5.1.5 Sharded Operation

Note: Changed in version 2.1.

Some aggregation operations using `aggregate` will cause `mongos` instances to require more CPU resources than in previous versions. This modified performance profile may dictate alternate architectural decisions if you use the *aggregation framework* extensively in a sharded environment.

The aggregation framework is compatible with sharded collections.

When operating on a sharded collection, the aggregation pipeline splits into two parts. The aggregation framework pushes all of the operators up to the first `$group` or `$sort` operation to each shard. [2] Then, a second pipeline on the `mongos` runs. This pipeline consists of the first `$group` or `$sort` and any remaining pipeline operators, and runs on the results received from the shards.

The `$group` operator brings in any "sub-totals" from the shards and combines them: in some cases these may be structures. For example, the `$avg` expression maintains a total and count for each shard; `mongos` combines these values and then divides.

5.1.6 Limitations

Aggregation operations with the `aggregate` command have the following limitations:

- The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, `CodeWScope`.
- Output from the *pipeline* can only contain 16 megabytes. If your result set exceeds this limit, the `aggregate` command produces an error.
- If any single aggregation operation consumes more than 10 percent of system RAM the operation will produce an error.

5.2 Aggregation Framework Examples

MongoDB provides flexible data aggregation functionality with the `aggregate` command. For additional information about aggregation consider the following resources:

- *Aggregation Framework* (page 151)
- *Aggregation Framework Reference* (page 163)
- `http://docs.mongodb.org/manual/reference/sql-aggregation-comparison`

This document provides a number of practical examples that display the capabilities of the aggregation framework. All examples use a publicly available data set of all zipcodes and populations in the United States.

5.2.1 Requirements

`mongod` and `mongo`, version 2.2 or later.

5.2.2 Aggregations using the Zip Code Data Set

To run you will need the zipcode data set. These data are available at: media.mongodb.org/zips.json[3]. Use `mongoimport` to load this data set into your `mongod` instance.

Data Model

Each document in this collection has the following form:

[2] If an early `$match` can exclude shards through the use of the shard key in the predicate, then these operators are only pushed to the relevant shards.

[3] http://media.mongodb.org/zips.json

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

In these documents:

- The `_id` field holds the zipcode as a string.
- The `city` field holds the city.
- The `state` field holds the two letter state abbreviation.
- The `pop` field holds the population.
- The `loc` field holds the location as a latitude longitude pair.

All of the following examples use the `aggregate()` helper in the `mongo` shell. `aggregate()` provides a wrapper around the `aggregate` database command. See the documentation for your `driver` for a more idiomatic interface for data aggregation operations.

States with Populations Over 10 Million

To return all states with a population greater than 10 million, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :
                         { _id : "$state",
                           totalPop : { $sum : "$pop" } } },
                       { $match : {totalPop : { $gte : 10*1000*1000 } } } )
```

Aggregations operations using the `aggregate()` helper, process all documents on the `zipcodes` collection. `aggregate()` connects a number of *pipeline* (page 152) operators, which define the aggregation process.

In the above example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates documents for each state.

  These new per-state documents have one field in addition the `_id` field: `totalPop` which is a generated field using the `$sum` operation to calculate the total value of all `pop` fields in the source documents.

  After the `$group` operation the documents in the pipeline resemble the following:

  ```
  {
    "_id" : "AK",
    "totalPop" : 550043
  }
  ```

- the `$match` operation filters these documents so that the only documents that remain are those where the value of `totalPop` is greater than or equal to 10 million.

  The `$match` operation does not alter the documents, which have the same format as the documents output by `$group`.

The equivalent *SQL* for this operation is:

```
SELECT state, SUM(pop) AS pop
       FROM zips
       GROUP BY state
       HAVING pop > (10*1000*1000)
```

Average City Population by State

To return the average populations for cities in each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group :
                         { _id : { state : "$state", city : "$city" },
                           pop : { $sum : "$pop" } } },
                       { $group :
                       { _id : "$_id.state",
                         avgCityPop : { $avg : "$pop" } } } )
```

Aggregations operations using the `aggregate()` helper, process all documents on the `zipcodes` collection. `aggregate()` a number of *pipeline* (page 152) operators that define the aggregation process.

In the above example, the pipeline passes all documents in the `zipcodes` collection through the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source document.

  After this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- the second `$group` operator collects documents by the `state` field and use the `$avg` expression to compute a value for the `avgCityPop` field.

The final output of this aggregation operation is:

```
{
  "_id" : "MN",
  "avgCityPop" : 5335
},
```

Largest and Smallest Cities by State

To return the smallest and largest cities by population for each state, use the following aggregation operation:

```
db.zipcodes.aggregate( { $group:
                         { _id: { state: "$state", city: "$city" },
                           pop: { $sum: "$pop" } } },
                       { $sort: { pop: 1 } },
                       { $group:
                         { _id : "$_id.state",
                           biggestCity:  { $last: "$_id.city" },
                           biggestPop:   { $last: "$pop" },
                           smallestCity: { $first: "$_id.city" },
                           smallestPop:  { $first: "$pop" } } },
```

```
                    // the following $project is optional, and
                    // modifies the output format.

                    { $project:
                      { _id: 0,
                        state: "$_id",
                        biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
                        smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } } )
```

Aggregations operations using the `aggregate()` helper, process all documents on the `zipcodes` collection. `aggregate()` a number of *pipeline* (page 152) operators that define the aggregation process.

All documents from the `zipcodes` collection pass into the pipeline, which consists of the following steps:

- the `$group` operator collects all documents and creates new documents for every combination of the `city` and `state` fields in the source documents.

  By specifying the value of `_id` as a sub-document that contains both fields, the operation preserves the `state` field for use later in the pipeline. The documents produced by this stage of the pipeline have a second field, `pop`, which uses the `$sum` operator to provide the total of the `pop` fields in the source document.

  At this stage in the pipeline, the documents resemble the following:

  ```
  {
    "_id" : {
      "state" : "CO",
      "city" : "EDGEWATER"
    },
    "pop" : 13154
  }
  ```

- `$sort` operator orders the documents in the pipeline based on the value of the `pop` field from smallest to largest. This operation does not alter the documents.

- the second `$group` operator collects the documents in the pipeline by the `state` field, which is a field inside the nested `_id` document.

  Within each per-state document this `$group` operator specifies four fields: Using the `$last` expression, the `$group` operator creates the `biggestcity` and `biggestpop` fields that store the city with the largest population and that population. Using the `$first` expression, the `$group` operator creates the `smallestcity` and `smallestpop` fields that store the city with the smallest population and that population.

  The documents, at this stage in the pipeline resemble the following:

  ```
  {
    "_id" : "WA",
    "biggestCity" : "SEATTLE",
    "biggestPop" : 520096,
    "smallestCity" : "BENGE",
    "smallestPop" : 2
  }
  ```

- The final operation is `$project`, which renames the `_id` field to `state` and moves the `biggestCity`, `biggestPop`, `smallestCity`, and `smallestPop` into `biggestCity` and `smallestCity` sub-documents.

The final output of this aggregation operation is:

```
{
  "state" : "RI",
  "biggestCity" : {
```

```
    "name" : "CRANSTON",
    "pop" : 176404
  },
  "smallestCity" : {
    "name" : "CLAYVILLE",
    "pop" : 45
  }
}
```

5.2.3 Aggregation with User Preference Data

Data Model

Consider a hypothetical sports club with a database that contains a `user` collection that tracks user's join dates, sport preferences, and stores these data in documents that resemble the following:

```
{
  _id : "jane",
  joined : ISODate("2011-03-02"),
  likes : ["golf", "racquetball"]
}
{
  _id : "joe",
  joined : ISODate("2012-07-02"),
  likes : ["tennis", "golf", "swimming"]
}
```

Normalize and Sort Documents

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the `users` collection. You might do this to normalize user names for processing.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

All documents from the `users` collection passes through the pipeline, which consists of the following operations:

- The `$project` operator:
  - creates a new field called `name`.
  - converts the value of the `_id` to upper case, with the `$toUpper` operator. Then the `$project` creates a new field, named `name` to hold this value.
  - suppresses the `id` field. `$project` will pass the `_id` field by default, unless explicitly suppressed.
- The `$sort` operator orders the results by the `name` field.

The results of the aggregation would resemble the following:

```
{
  "name" : "JANE"
},
{
```

```
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

Return Usernames Ordered by Join Month

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

```
db.users.aggregate(
  [
    { $project : { month_joined : {
                                      $month : "$joined"
                                    },
                   name : "$_id",
                   _id : 0
                 },
    { $sort : { month_joined : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator:
  - Creates two new fields: `month_joined` and `name`.
  - Suppresses the `id` from the results. The `aggregate()` method includes the `_id`, unless explicitly suppressed.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns those values to the `month_joined` field.
- The `$sort` operator sorts the results by the `month_joined` field.

The operation returns results that resemble the following:

```
{
  "month_joined" : 1,
  "name" : "ruth"
},
{
  "month_joined" : 1,
  "name" : "harold"
},
{
  "month_joined" : 1,
  "name" : "kate"
}
{
  "month_joined" : 2,
  "name" : "jill"
}
```

Return Total Number of Joins per Month

The following operation shows how many people joined each month of the year. You might use this aggregated data for such information for recruiting and marketing strategies.

```
db.users.aggregate(
  [
    { $project : { month_joined : { $month : "$joined" } } } ,
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum : 1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

The pipeline passes all documents in the `users` collection through the following operations:

- The `$project` operator creates a new field called `month_joined`.
- The `$month` operator converts the values of the `joined` field to integer representations of the month. Then the `$project` operator assigns the values to the `month_joined` field.
- The `$group` operator collects all documents with a given `month_joined` value and counts how many documents there are for that value. Specifically, for each unique value, `$group` creates a new "per-month" document with two fields:
  - `_id`, which contains a nested document with the `month_joined` field and its value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `month_joined` value.
- The `$sort` operator sorts the documents created by `$group` according to the contents of the `month_joined` field.

The result of this aggregation operation would resemble the following:

```
{
  "_id" : {
    "month_joined" : 1
  },
  "number" : 3
},
{
  "_id" : {
    "month_joined" : 2
  },
  "number" : 9
},
{
  "_id" : {
    "month_joined" : 3
  },
  "number" : 5
}
```

Return the Five Most Common "Likes"

The following aggregation collects top five most "liked" activities in the data set. In this data set, you might use an analysis of this to help inform planning and future development.

```
db.users.aggregate(
  [
    { $unwind : "$likes" },
    { $group : { _id : "$likes" , number : { $sum : 1 } } },
    { $sort : { number : -1 } },
    { $limit : 5 }
  ]
)
```

The pipeline begins with all documents in the `users` collection, and passes these documents through the following operations:

- The `$unwind` operator separates each value in the `likes` array, and creates a new version of the source document for every element in the array.

  Example

  Given the following document from the `users` collection:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : ["golf", "racquetball"]
  }
  ```

  The `$unwind` operator would create the following documents:

  ```
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : "golf"
  }
  {
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : "racquetball"
  }
  ```

- The `$group` operator collects all documents the same value for the `likes` field and counts each grouping. With this information, `$group` creates a new document with two fields:
  - `_id`, which contains the `likes` value.
  - `number`, which is a generated field. The `$sum` operator increments this field by 1 for every document containing the given `likes` value.
- The `$sort` operator sorts these documents by the `number` field in reverse order.
- The `$limit` operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

```
{
  "_id" : "golf",
  "number" : 33
},
{
  "_id" : "racquetball",
  "number" : 31
},
```

```
{
  "_id" : "swimming",
  "number" : 24
},
{
  "_id" : "handball",
  "number" : 19
},
{
  "_id" : "tennis",
  "number" : 18
}
```

5.3 Aggregation Framework Reference

New in version 2.1.0.

The aggregation framework provides the ability to project, process, and/or control the output of the query, without using *map-reduce*. Aggregation uses a syntax that resembles the same syntax and form as "regular" MongoDB database queries.

These aggregation operations are all accessible by way of the `aggregate()` method. While all examples in this document use this method, `aggregate()` is merely a wrapper around the *database command* `aggregate`. The following prototype aggregation operations are equivalent:

```
db.people.aggregate( <pipeline> )
db.people.aggregate( [<pipeline>] )
db.runCommand( { aggregate: "people", pipeline: [<pipeline>] } )
```

These operations perform aggregation routines on the collection named `people`. `<pipeline>` is a placeholder for the aggregation *pipeline* definition. `aggregate()` accepts the stages of the pipeline (i.e. `<pipeline>`) as an array, or as arguments to the method.

This documentation provides an overview of all aggregation operators available for use in the aggregation pipeline as well as details regarding their use and behavior.

See also:

Aggregation Framework (page 151) overview, the *Aggregation Framework Documentation Index* (page 151), and the *Aggregation Framework Examples* (page 155) for more information on the aggregation functionality.

Aggregation Operators:

5.3.1 Pipeline

Warning: The pipeline cannot operate on values of the following types: `Binary`, `Symbol`, `MinKey`, `MaxKey`, `DBRef`, `Code`, and `CodeWScope`.

Pipeline operators appear in an array. Conceptually, documents pass through these operators in a sequence. All examples in this section assume that the aggregation pipeline begins with a collection named `article` that contains documents that resemble the following:

```
{
  title : "this is my title" ,
  author : "bob" ,
  posted : new Date() ,
  pageViews : 5 ,
  tags : [ "fun" , "good" , "fun" ] ,
  comments : [
      { author :"joe" , text : "this is cool" } ,
      { author :"sam" , text : "this is bad" }
  ],
  other : { foo : 5 }
}
```

The current pipeline operators are:

`$project`

Reshapes a document stream by renaming, adding, or removing fields. Also use `$project` to create computed values or sub-documents. Use `$project` to:

- Include fields from the original document.
- Insert computed fields.
- Rename fields.
- Create and populate fields that hold sub-documents.

Use `$project` to quickly select the fields that you want to include or exclude from the response. Consider the following aggregation framework operation.

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        author : 1 ,
    }}
 );
```

This operation includes the `title` field and the `author` field in the document that returns from the aggregation *pipeline*.

Note: The `_id` field is always included by default. You may explicitly exclude `_id` as follows:

```
db.article.aggregate(
    { $project : {
        _id : 0 ,
        title : 1 ,
        author : 1
    }}
);
```

Here, the projection excludes the `_id` field but includes the `title` and `author` fields.

Projections can also add computed fields to the document stream passing through the pipeline. A computed field can use any of the *expression operators* (page 170). Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1,
        doctoredPageViews : { $add:["$pageViews", 10] }
    }}
);
```

Here, the field `doctoredPageViews` represents the value of the `pageViews` field after adding 10 to the original field using the `$add`.

Note: You must enclose the expression that defines the computed field in braces, so that the expression is a valid object.

You may also use `$project` to rename fields. Consider the following example:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        page_views : "$pageViews" ,
        bar : "$other.foo"
    }}
);
```

This operation renames the `pageViews` field to `page_views`, and renames the `foo` field in the `other` sub-document as the top-level field `bar`. The field references used for renaming fields are direct expressions and do not use an operator or surrounding braces. All aggregation field references can use dotted paths to refer to fields in nested documents.

Finally, you can use the `$project` to create and populate new sub-documents. Consider the following example that creates a new object-valued field named `stats` that holds a number of values:

```
db.article.aggregate(
    { $project : {
        title : 1 ,
        stats : {
            pv : "$pageViews",
            foo : "$other.foo",
            dpv : { $add:["$pageViews", 10] }
        }
    }}
);
```

This projection includes the `title` field and places `$project` into "inclusive" mode. Then, it creates the `stats` documents with the following fields:

- `pv` which includes and renames the `pageViews` from the top level of the original documents.
- `foo` which includes the value of `other.foo` from the original documents.
- `dpv` which is a computed field that adds 10 to the value of the `pageViews` field in the original document using the `$add` aggregation expression.

`$match`

Provides a query-like interface to filter documents out of the aggregation *pipeline*. The `$match` drops documents that do not match the condition from the aggregation pipeline, and it passes documents that match along the pipeline unaltered.

The syntax passed to the `$match` is identical to the *query* syntax. Consider the following prototype form:

```
db.article.aggregate(
    { $match : <match-predicate> }
);
```

The following example performs a simple field equality test:

```
db.article.aggregate(
    { $match : { author : "dave" } }
);
```

This operation only returns documents where the `author` field holds the value `dave`. Consider the following example, which performs a range test:

```
db.article.aggregate(
    { $match : { score  : { $gt : 50, $lte : 90 } } }
);
```

Here, all documents return when the `score` field holds a value that is greater than 50 and less than or equal to 90.

Note: Place the `$match` as early in the aggregation *pipeline* as possible. Because `$match` limits the total number of documents in the aggregation pipeline, earlier `$match` operations minimize the amount of later processing. If you place a `$match` at the very beginning of a pipeline, the query can take advantage of *indexes* like any other `db.collection.find()` or `db.collection.findOne()`.

Warning: You cannot use `$where` or *geospatial operations* in `$match` queries as part of the aggregation pipeline.

`$limit`

Restricts the number of *documents* that pass through the `$limit` in the *pipeline*.

`$limit` takes a single numeric (positive whole number) value as a parameter. Once the specified number of documents pass through the pipeline operator, no more will. Consider the following example:

```
db.article.aggregate(
    { $limit : 5 }
);
```

This operation returns only the first 5 documents passed to it from by the pipeline. `$limit` has no effect on the content of the documents it passes.

`$skip`

Skips over the specified number of *documents* that pass through the `$skip` in the *pipeline* before passing all of the remaining input.

`$skip` takes a single numeric (positive whole number) value as a parameter. Once the operation has skipped the specified number of documents, it passes all the remaining documents along the *pipeline* without alteration. Consider the following example:

```
db.article.aggregate(
    { $skip : 5 }
);
```

This operation skips the first 5 documents passed to it by the pipeline. `$skip` has no effect on the content of the documents it passes along the pipeline.

$unwind

Peels off the elements of an array individually, and returns a stream of documents. `$unwind` returns one document for every member of the unwound array within every source document. Take the following aggregation command:

```
db.article.aggregate(
    { $project : {
        author : 1 ,
        title : 1 ,
        tags : 1
    }},
    { $unwind : "$tags" }
);
```

Note: The dollar sign (i.e. `$`) must proceed the field specification handed to the `$unwind` operator.

In the above aggregation `$project` selects (inclusively) the `author`, `title`, and `tags` fields, as well as the `_id` field implicitly. Then the pipeline passes the results of the projection to the `$unwind` operator, which will unwind the `tags` field. This operation may return a sequence of documents that resemble the following for a collection that contains one document holding a `tags` field with an array of 3 items.

```
{
     "result" : [
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "good"
             },
             {
                     "_id" : ObjectId("4e6e4ef557b77501a49233f6"),
                     "title" : "this is my title",
                     "author" : "bob",
                     "tags" : "fun"
             }
     ],
     "OK" : 1
}
```

A single document becomes 3 documents: each document is identical except for the value of the `tags` field. Each value of `tags` is one of the values in the original "tags" array.

Note: `$unwind` has the following behaviors:

- `$unwind` is most useful in combination with `$group`.
- You may undo the effects of unwind operation with the `$group` pipeline operator.
- If you specify a target field for `$unwind` that does not exist in an input document, the pipeline ignores the input document, and will generate no result documents.
- If you specify a target field for `$unwind` that is not an array, `db.collection.aggregate()` generates an error.

- If you specify a target field for `$unwind` that holds an empty array (`[]`) in an input document, the pipeline ignores the input document, and will generates no result documents.

`$group`

Groups documents together for the purpose of calculating aggregate values based on a collection of documents. Practically, group often supports tasks such as average page views for each page in a website on a daily basis.

The output of `$group` depends on how you define groups. Begin by specifying an identifier (i.e. a `_id` field) for the group you're creating with this pipeline. You can specify a single field from the documents in the pipeline, a previously computed value, or an aggregate key made up from several incoming fields. Aggregate keys may resemble the following document:

```
{ _id : { author: '$author', pageViews: '$pageViews', posted: '$posted' } }
```

With the exception of the `_id` field, `$group` cannot output nested documents.

Important: The output of `$group` is not ordered.

Every group expression must specify an `_id` field. You may specify the `_id` field as a dotted field path reference, a document with multiple fields enclosed in braces (i.e. `{` and `}`), or a constant value.

Note: Use `$project` as needed to rename the grouped field after an `$group` operation, if necessary.

Consider the following example:

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

This groups by the `author` field and computes two fields, the first `docsPerAuthor` is a counter field that adds one for each document with a given author field using the `$sum` function. The `viewsPerAuthor` field is the sum of all of the `pageViews` fields in the documents for each group.

Each field defined for the `$group` must use one of the group aggregation function listed below to generate its composite value:

- `$addToSet`
- `$first`
- `$last`
- `$max`
- `$min`
- `$avg`
- `$push`
- `$sum`

Warning: The aggregation system currently stores `$group` operations in memory, which may cause problems when processing a larger number of groups.

$sort

The $sort *pipeline* operator sorts all input documents and returns them to the pipeline in sorted order. Consider the following prototype form:

```
db.<collection-name>.aggregate(
    { $sort : { <sort-key> } }
);
```

This sorts the documents in the collection named <collection-name>, according to the key and specification in the { <sort-key> } document.

Specify the sort in a document with a field or fields that you want to sort by and a value of 1 or -1 to specify an ascending or descending sort respectively, as in the following example:

```
db.users.aggregate(
    { $sort : { age : -1, posts: 1 } }
);
```

This operation sorts the documents in the users collection, in descending order according by the age field and then in ascending order according to the value in the posts field.

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

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

Note: The $sort cannot begin sorting documents until previous operators in the pipeline have returned all output.

- $skip

$sort operator can take advantage of an index when placed at the **beginning** of the pipeline or placed **before** the following aggregation operators:

- $project
- $unwind
- $group.

Warning: Unless the `$sort` operator can use an index, in the current release, the sort must fit within memory. This may cause problems when sorting large numbers of documents.

5.3.2 Expressions

These operators calculate values within the *aggregation framework*.

`$group` Operators

The `$group` pipeline stage provides the following operations:

`$addToSet`
Returns an array of all the values found in the selected field among the documents in that group. *Every unique value only appears once* in the result set. There is no ordering guarantee for the output documents.

`$first`
Returns the first value it encounters for its group .

Note: Only use `$first` when the `$group` follows an `$sort` operation. Otherwise, the result of this operation is unpredictable.

`$last`
Returns the last value it encounters for its group.

Note: Only use `$last` when the `$group` follows an `$sort` operation. Otherwise, the result of this operation is unpredictable.

`$max`
Returns the highest value among all values of the field in all documents selected by this group.

`$min`
Returns the lowest value among all values of the field in all documents selected by this group.

`$avg`
Returns the average of all the values of the field in all documents selected by this group.

`$push`
Returns an array of all the values found in the selected field among the documents in that group. *A value may appear more than once* in the result set if more than one field in the grouped documents has that value.

`$sum`
Returns the sum of all the values for a specified field in the grouped documents, as in the second use above.

Alternately, if you specify a value as an argument, `$sum` will increment this field by the specified value for every document in the grouping. Typically, as in the first use above, specify a value of `1` in order to count members of the group.

Boolean Operators

The three boolean operators accept Booleans as arguments and return Booleans as results.

Note: These operators convert non-booleans to Boolean values according to the BSON standards. Here, `null`, `undefined`, and `0` values become `false`, while non-zero numeric values, and all other types, such as strings, dates, objects become `true`.

$and

Takes an array one or more values and returns `true` if *all* of the values in the array are `true`. Otherwise `$and` returns `false`.

Note: `$and` uses short-circuit logic: the operation stops evaluation after encountering the first `false` expression.

$or

Takes an array of one or more values and returns `true` if *any* of the values in the array are `true`. Otherwise `$or` returns false.

Note: `$or` uses short-circuit logic: the operation stops evaluation after encountering the first `true` expression.

$not

Returns the boolean opposite value passed to it. When passed a `true` value, `$not` returns `false`; when passed a `false` value, `$not` returns `true`.

Comparison Operators

These operators perform comparisons between two values and return a Boolean, in most cases, reflecting the result of that comparison.

All comparison operators take an array with a pair of values. You may compare numbers, strings, and dates. Except for `$cmp`, all comparison operators return a Boolean value. `$cmp` returns an integer.

$cmp

Takes two values in an array and returns an integer. The returned value is:

- A negative number if the first value is less than the second.
- A positive number if the first value is greater than the second.
- `0` if the two values are equal.

$eq

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the values are equivalent.
- `false` when the values are **not** equivalent.

$gt

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the first value is *greater than* the second value.
- `false` when the first value is *less than or equal to* the second value.

$gte

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the first value is *greater than or equal* to the second value.
- `false` when the first value is *less than* the second value.

$lt

Takes two values in an array and returns a boolean. The returned value is:

- `true` when the first value is *less than* the second value.

- `false` when the first value is *greater than or equal to* the second value.

`$lte`
Takes two values in an array and returns a boolean. The returned value is:

- `true` when the first value is *less than or equal to* the second value.
- `false` when the first value is *greater than* the second value.

`$ne`
Takes two values in an array returns a boolean. The returned value is:

- `true` when the values are **not equivalent**.
- `false` when the values are **equivalent**.

Arithmetic Operators

These operators only support numbers.

`$add`
Takes an array of one or more numbers and adds them together, returning the sum.

`$divide`
Takes an array that contains a pair of numbers and returns the value of the first number divided by the second number.

`$mod`
Takes an array that contains a pair of numbers and returns the *remainder* of the first number divided by the second number.

See also:

`$mod`

`$multiply`
Takes an array of one or more numbers and multiples them, returning the resulting product.

`$subtract`
Takes an array that contains a pair of numbers and subtracts the second from the first, returning their difference.

String Operators

These operators manipulate strings within projection expressions.

`$strcasecmp`
Takes in two strings. Returns a number. `$strcasecmp` is positive if the first string is "greater than" the second and negative if the first string is "less than" the second. `$strcasecmp` returns 0 if the strings are identical.

Note: `$strcasecmp` may not make sense when applied to glyphs outside the Roman alphabet.

`$strcasecmp` internally capitalizes strings before comparing them to provide a case-*insensitive* comparison. Use `$cmp` for a case sensitive comparison.

`$substr`
`$substr` takes a string and two numbers. The first number represents the number of bytes in the string to skip, and the second number specifies the number of bytes to return from the string.

Note: `$substr` is not encoding aware and if used improperly may produce a result string containing an invalid UTF-8 character sequence.

$toLower
Takes a single string and converts that string to lowercase, returning the result. All uppercase letters become lowercase.

Note: `$toLower` may not make sense when applied to glyphs outside the Roman alphabet.

$toUpper
Takes a single string and converts that string to uppercase, returning the result. All lowercase letters become uppercase.

Note: `$toUpper` may not make sense when applied to glyphs outside the Roman alphabet.

Date Operators

All date operators take a "Date" typed value as a single argument and return a number.

$dayOfYear
Takes a date and returns the day of the year as a number between 1 and 366.

$dayOfMonth
Takes a date and returns the day of the month as a number between 1 and 31.

$dayOfWeek
Takes a date and returns the day of the week as a number between 1 (Sunday) and 7 (Saturday.)

$year
Takes a date and returns the full year.

$month
Takes a date and returns the month as a number between 1 and 12.

$week
Takes a date and returns the week of the year as a number between 0 and 53.

Weeks begin on Sundays, and week 1 begins with the first Sunday of the year. Days preceding the first Sunday of the year are in week 0. This behavior is the same as the "`%U`" operator to the `strftime` standard library function.

$hour
Takes a date and returns the hour between 0 and 23.

$minute
Takes a date and returns the minute between 0 and 59.

$second
Takes a date and returns the second between 0 and 59, but can be 60 to account for leap seconds.

Conditional Expressions

$cond
Use the `$cond` operator with the following syntax:

```
{ $cond: [ <boolean-expression>, <true-case>, <false-case> ] }
```

Takes an array with three expressions, where the first expression evaluates to a Boolean value. If the first expression evaluates to true, `$cond` returns the value of the second expression. If the first expression evaluates to false, `$cond` evaluates and returns the third expression.

`$ifNull`

Use the `$ifNull` operator with the following syntax:

```
{ $ifNull: [ <expression>, <replacement-if-null> ] }
```

Takes an array with two expressions. `$ifNull` returns the first expression if it evaluates to a non-null value. Otherwise, `$ifNull` returns the second expression's value.

5.4 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.

For many simple aggregation tasks, see the *aggregation framework* (page 151).

5.4.1 Map-Reduce Examples

This section provides some map-reduce examples in the `mongo` shell using the `db.collection.mapReduce()` method:

```
db.collection.mapReduce(
                         <mapfunction>,
                         <reducefunction>,
                         {
                           out: <collection>,
                           query: <document>,
                           sort: <document>,
                           limit: <number>,
                           finalize: <function>,
                           scope: <document>,
                           jsMode: <boolean>,
                           verbose: <boolean>
                         }
                       )
```

For more information on the parameters, see the `db.collection.mapReduce()` reference page .

Consider the following map-reduce operations on a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 25,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

Return the Total Price Per Customer Id

Perform map-reduce operation on the `orders` collection to group by the `cust_id`, and for each `cust_id`, calculate the sum of the `price` for each `cust_id`:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - The function maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair.

```
var mapFunction1 = function() {
                       emit(this.cust_id, this.price);
                   };
```

2. Define the corresponding reduce function with two arguments `keyCustId` and `valuesPrices`:
   - The `valuesPrices` is an array whose elements are the `price` values emitted by the map function and grouped by `keyCustId`.
   - The function reduces the `valuesPrice` array to the sum of its elements.

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                          return Array.sum(valuesPrices);
                      };
```

3. Perform the map-reduce on all documents in the `orders` collection using the `mapFunction1` map function and the `reduceFunction1` reduce function.

```
db.orders.mapReduce(
                     mapFunction1,
                     reduceFunction1,
                     { out: "map_reduce_example" }
                   )
```

   This operation outputs the results to a collection named `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will replace the contents with the results of this map-reduce operation:

Calculate the Number of Orders, Total Quantity, and Average Quantity Per Item

In this example you will perform a map-reduce operation on the `orders` collection, for all documents that have an `ord_date` value greater than `01/01/2012`. The operation groups by the `item.sku` field, and for each `sku` calculates the number of orders and the total quantity ordered. The operation concludes by calculating the average quantity per order for each `sku` value:

1. Define the map function to process each input document:
   - In the function, `this` refers to the document that the map-reduce operation is processing.
   - For each item, the function associates the `sku` with a new object `value` that contains the `count` of `1` and the item `qty` for the order and emits the `sku` and `value` pair.

```
var mapFunction2 = function() {
                       for (var idx = 0; idx < this.items.length; idx++) {
                           var key = this.items[idx].sku;
                           var value = {
                                         count: 1,
```

```
                                        qty: this.items[idx].qty
                                    };
                emit(key, value);
            }
        };
```

2. Define the corresponding reduce function with two arguments `keySKU` and `valuesCountObjects`:

   - `valuesCountObjects` 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 `valuesCountObjects` array to a single object `reducedValue` that also contains the `count` and the `qty` fields.
   - In `reducedValue`, the `count` field contains the sum of the `count` fields from the individual array elements, and the `qty` field contains the sum of the `qty` fields from the individual array elements.

```
var reduceFunction2 = function(keySKU, 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. Define a finalize function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` object to add a computed field named `average` and returns the modified object:

```
var finalizeFunction2 = function (key, reducedValue) {

                            reducedValue.average = reducedValue.qty/reducedValue.count;

                            return reducedValue;
                        };
```

4. Perform the map-reduce operation on the `orders` collection using the `mapFunction2`, `reduceFunction2`, and `finalizeFunction2` functions.

```
db.orders.mapReduce( mapFunction2,
                     reduceFunction2,
                     {
                       out: { merge: "map_reduce_example" },
                       query: { ord_date: { $gt: new Date('01/01/2012') } },
                       finalize: finalizeFunction2
                     }
                   )
```

   This operation uses the `query` field to select only those documents with `ord_date` greater than `new Date(01/01/2012)`. Then it output the results to a collection `map_reduce_example`. If the `map_reduce_example` collection already exists, the operation will merge the existing contents with the results of this map-reduce operation:

5.4.2 Incremental Map-Reduce

If the map-reduce dataset is constantly growing, then rather than performing the map-reduce operation over the entire dataset each time you want to run map-reduce, you may want to perform an incremental map-reduce.

To perform incremental map-reduce:

1. Run a map-reduce job over the current collection and output the result to a separate collection.
2. When you have more data to process, run subsequent map-reduce job with:
   - the `query` parameter that specifies conditions that match *only* the new documents.
   - the `out` parameter that specifies the `reduce` action to merge the new results into the existing output collection.

Consider the following example where you schedule a map-reduce operation on a `sessions` collection to run at the end of each day.

Data Setup

The `sessions` collection contains documents that log users' session each day, for example:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-03 14:17:00'), length: 95 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-03 14:23:00'), length: 110 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-03 15:02:00'), length: 120 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-03 16:45:00'), length: 45 } );

db.sessions.save( { userid: "a", ts: ISODate('2011-11-04 11:05:00'), length: 105 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-04 13:14:00'), length: 120 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-04 17:00:00'), length: 130 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-04 15:37:00'), length: 65 } );
```

Initial Map-Reduce of Current Collection

Run the first map-reduce operation as follows:

1. Define the `map` function that maps the `userid` to an object that contains the fields `userid`, `total_time`, `count`, and `avg_time`:

   ```
   var mapFunction = function() {
                         var key = this.userid;
                         var value = {
                                       userid: this.userid,
                                       total_time: this.length,
                                       count: 1,
                                       avg_time: 0
                                      };

                         emit( key, value );
                     };
   ```

2. Define the corresponding `reduce` function with two arguments `key` and `values` to calculate the total time and the count. The `key` corresponds to the `userid`, and the `values` is an array whose elements corresponds to the individual objects mapped to the `userid` in the `mapFunction`.

   ```
   var reduceFunction = function(key, values) {

                           var reducedObject = {
                                                 userid: key,
                                                 total_time: 0,
                                                 count:0,
                                                 avg_time:0
                                               };
   ```

```
                values.forEach( function(value) {
                                        reducedObject.total_time += value.total_time;
                                        reducedObject.count += value.count;
                                    }
                                );
                return reducedObject;
            };
```

3. Define `finalize` function with two arguments `key` and `reducedValue`. The function modifies the `reducedValue` document to add another field `average` and returns the modified document.

```
var finalizeFunction = function (key, reducedValue) {

                if (reducedValue.count > 0)
                    reducedValue.avg_time = reducedValue.total_time / reducedValue.cou

                return reducedValue;
            };
```

4. Perform map-reduce on the `session` collection using the `mapFunction`, the `reduceFunction`, and the `finalizeFunction` functions. Output the results to a collection `session_stat`. If the `session_stat` collection already exists, the operation will replace the contents:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         out: { reduce: "session_stat" },
                         finalize: finalizeFunction
                       }
                     )
```

Subsequent Incremental Map-Reduce

Later as the `sessions` collection grows, you can run additional map-reduce operations. For example, add new documents to the `sessions` collection:

```
db.sessions.save( { userid: "a", ts: ISODate('2011-11-05 14:17:00'), length: 100 } );
db.sessions.save( { userid: "b", ts: ISODate('2011-11-05 14:23:00'), length: 115 } );
db.sessions.save( { userid: "c", ts: ISODate('2011-11-05 15:02:00'), length: 125 } );
db.sessions.save( { userid: "d", ts: ISODate('2011-11-05 16:45:00'), length: 55 } );
```

At the end of the day, perform incremental map-reduce on the `sessions` collection but use the `query` field to select only the new documents. Output the results to the collection `session_stat`, but `reduce` the contents with the results of the incremental map-reduce:

```
db.sessions.mapReduce( mapFunction,
                       reduceFunction,
                       {
                         query: { ts: { $gt: ISODate('2011-11-05 00:00:00') } },
                         out: { reduce: "session_stat" },
                         finalize: finalizeFunction
                       }
                     );
```

5.4.3 Temporary Collection

The map-reduce operation uses a temporary collection during processing. At completion, the map-reduce operation renames the temporary collection. As a result, you can perform a map-reduce operation periodically with the same target collection name without affecting the intermediate states. Use this mode when generating statistical output collections on a regular basis.

5.4.4 Concurrency

The map-reduce operation is composed of many tasks, including:

- reads from the input collection,
- executions of the `map` function,
- executions of the `reduce` function,
- writes to the output collection.

These various tasks take the following locks:

- The read phase takes a read lock. It yields every 100 documents.
- The JavaScript code (i.e. `map`, `reduce`, `finalize` functions) is executed in a single thread, taking a JavaScript lock; however, most JavaScript tasks in map-reduce are very short and yield the lock frequently.
- 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.

Although single-threaded, the map-reduce tasks interleave and appear to run in parallel.

Note: The final write lock during post-processing makes the results appear atomically. However, output actions `merge` and `reduce` may take minutes to process. For the `merge` and `reduce`, the `nonAtomic` flag is available. See the `db.collection.mapReduce()` reference for more information.

5.4.5 Sharded Cluster

Sharded 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 Output

By default the output collection is not sharded. The process is:

- `mongos` dispatches a map-reduce finish job to the shard that will store the target collection.
- The target shard pulls results from all other shards, and runs a final reduce/finalize operation, and write to the output.

- If using the `sharded` option to the `out` parameter, MongoDB shards the output using `_id` field as the shard key.

  Changed in version 2.2.

- If the output collection does not exist, MongoDB creates and shards the collection on the `_id` field. If the collection is empty, MongoDB creates *chunks* using the result of the first stage of the map-reduce operation.
- `mongos` dispatches, in parallel, a map-reduce finish job to every shard that owns a chunk.
- Each shard will pull the results it owns from all other shards, run a final reduce/finalize, and write to the output collection.

Note:

- During later map-reduce jobs, MongoDB splits chunks as needed.
- Balancing of chunks for the output collection is automatically prevented during post-processing to avoid concurrency issues.

In MongoDB 2.0:

- `mongos` retrieves the results from each shard, and performs merge sort to order the results, and performs a reduce/finalize 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 datasets.
- Shard chunks are not automatically split during insertion. This requires manual intervention until the chunks are granular and balanced.

Warning: For best results, only use the sharded output options for `mapReduce` in version 2.2 or later.

5.4.6 Troubleshooting Map-Reduce Operations

You can troubleshoot the `map` function and the `reduce` function in the `mongo` shell.

Troubleshoot the Map Function

You can verify the `key` and `value` pairs emitted by the `map` function by writing your own `emit` function.

Consider a collection `orders` that contains documents of the following prototype:

```
{
     _id: ObjectId("50a8240b927d5d8b5891743c"),
     cust_id: "abc123",
     ord_date: new Date("Oct 04, 2012"),
     status: 'A',
     price: 250,
     items: [ { sku: "mmm", qty: 5, price: 2.5 },
              { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

1. Define the `map` function that maps the `price` to the `cust_id` for each document and emits the `cust_id` and `price` pair:

   ```
   var map = function() {
       emit(this.cust_id, this.price);
   };
   ```

2. Define the `emit` function to print the key and value:

```
var emit = function(key, value) {
    print("emit");
    print("key: " + key + "  value: " + tojson(value));
}
```

3. Invoke the `map` function with a single document from the `orders` collection:

```
var myDoc = db.orders.findOne( { _id: ObjectId("50a8240b927d5d8b5891743c") } );
map.apply(myDoc);
```

4. Verify the key and value pair is as you expected.

```
emit
key: abc123 value:250
```

5. Invoke the `map` function with multiple documents from the `orders` collection:

```
var myCursor = db.orders.find( { cust_id: "abc123" } );

while (myCursor.hasNext()) {
    var doc = myCursor.next();
    print ("document _id= " + tojson(doc._id));
    map.apply(doc);
    print();
}
```

6. Verify the key and value pairs are as you expected.

Troubleshoot the Reduce Function

Confirm Output Type

You can test that the `reduce` function returns a value that is the same type as the value emitted from the `map` function.

1. Define a `reduceFunction1` function that takes the arguments `keyCustId` and `valuesPrices`. `valuesPrices` is an array of integers:

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
                          return Array.sum(valuesPrices);
                      };
```

2. Define a sample array of integers:

```
var myTestValues = [ 5, 5, 10 ];
```

3. Invoke the `reduceFunction1` with `myTestValues`:

```
reduceFunction1('myKey', myTestValues);
```

4. Verify the `reduceFunction1` returned an integer:

```
20
```

5. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

6. Define a sample array of documents:

```
var myTestObjects = [
                        { count: 1, qty: 5 },
                        { count: 2, qty: 10 },
                        { count: 3, qty: 15 }
                    ];
```

7. Invoke the `reduceFunction2` with `myTestObjects`:

```
reduceFunction2('myKey', myTestObjects);
```

8. Verify the `reduceFunction2` returned a document with exactly the `count` and the `qty` field:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Insensitivity to the Order of Mapped Values

The `reduce` function takes a `key` and a `values` array as its argument. You can test that the result of the `reduce` function does not depend on the order of the elements in the `values` array.

1. Define a sample `values1` array and a sample `values2` array that only differ in the order of the array elements:

```
var values1 = [
                  { count: 1, qty: 5 },
                  { count: 2, qty: 10 },
                  { count: 3, qty: 15 }
                ];

var values2 = [
                  { count: 3, qty: 15 },
                  { count: 1, qty: 5 },
                  { count: 2, qty: 10 }
                ];
```

2. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }
```

```
            return reducedValue;
        };
```

3. Invoke the `reduceFunction2` first with `values1` and then with `values2`:

```
reduceFunction2('myKey', values1);
reduceFunction2('myKey', values2);
```

4. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

Ensure Reduce Function Idempotentcy

Because the map-reduce operation may call a `reduce` multiple times for the same key, the `reduce` function must return a value of the same type as the value emitted from the `map` function. You can test that the `reduce` function process "reduced" values without affecting the *final* value.

1. Define a `reduceFunction2` function that takes the arguments `keySKU` and `valuesCountObjects`. `valuesCountObjects` is an array of documents that contain two fields `count` and `qty`:

```
var reduceFunction2 = function(keySKU, valuesCountObjects) {
                          reducedValue = { count: 0, qty: 0 };

                          for (var idx = 0; idx < valuesCountObjects.length; idx++) {
                              reducedValue.count += valuesCountObjects[idx].count;
                              reducedValue.qty += valuesCountObjects[idx].qty;
                          }

                          return reducedValue;
                      };
```

2. Define a sample key:

```
var myKey = 'myKey';
```

3. Define a sample `valuesIdempotent` array that contains an element that is a call to the `reduceFunction2` function:

```
var valuesIdempotent = [
                         { count: 1, qty: 5 },
                         { count: 2, qty: 10 },
                         reduceFunction2(myKey, [ { count:3, qty: 15 } ] )
                       ];
```

4. Define a sample `values1` array that combines the values passed to `reduceFunction2`:

```
var values1 = [
                { count: 1, qty: 5 },
                { count: 2, qty: 10 },
                { count: 3, qty: 15 }
              ];
```

5. Invoke the `reduceFunction2` first with `myKey` and `valuesIdempotent` and then with `myKey` and `values1`:

```
reduceFunction2(myKey, valuesIdempotent);
reduceFunction2(myKey, values1);
```

6. Verify the `reduceFunction2` returned the same result:

```
{ "count" : 6, "qty" : 30 }
```

In addition to the aggregation framework, MongoDB provides simple *aggregation methods and commands* (page 184), that you may find useful for some classes of tasks:

5.5 Simple Aggregation Methods and Commands

In addition to the *aggregation framework* (page 151) and *map-reduce*, MongoDB provides the following methods and commands to perform aggregation:

5.5.1 Count

MongoDB offers the following command and methods to provide `count` functionality:

- `http://docs.mongodb.org/manual/reference/command/count`
- `http://docs.mongodb.org/manual/reference/method/db.collection.count`
- `http://docs.mongodb.org/manual/reference/method/cursor.count`

5.5.2 Distinct

MongoDB offers the following command and method to provide the `distinct` functionality:

- `http://docs.mongodb.org/manual/reference/command/distinct`
- `http://docs.mongodb.org/manual/reference/method/db.collection.distinct`

5.5.3 Group

MongoDB offers the following command and method to provide `group` functionality:

- `http://docs.mongodb.org/manual/reference/command/group`
- `http://docs.mongodb.org/manual/reference/method/db.collection.group`

CHAPTER 6

Indexes

Indexes provide high performance read operations for frequently used queries. Indexes are particularly useful where the total size of the documents exceeds the amount of available RAM.

For basic concepts and options, see *Indexing Overview* (page 185). For procedures and operational concerns, see *Indexing Operations* (page 194). For information on how applications might use indexes, see *Indexing Strategies* (page 199).

The following outlines the indexing documentation:

6.1 Indexing Overview

This document provides an overview of indexes in MongoDB, including index types and creation options. For operational guidelines and procedures, see the *Indexing Operations* (page 194) document. For strategies and practical approaches, see the *Indexing Strategies* (page 199) document.

6.1.1 Synopsis

An index is a data structure that allows you to quickly locate documents based on the values stored in certain specified fields. Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB supports indexes on any field or sub-field contained in documents within a MongoDB collection.

MongoDB indexes have the following core features:

- MongoDB defines indexes on a per-*collection* level.
- You can create indexes on a single field or on multiple fields using a *compound index* (page 187).
- Indexes enhance query performance, often dramatically. However, each index also incurs some overhead for every write operation. Consider the queries, the frequency of these queries, the size of your working set, the insert load, and your application's requirements as you create indexes in your MongoDB environment.
- All MongoDB indexes use a B-tree data structure. MongoDB can use these representation of the data to optimize query responses.
- Every query, including update operations, use one and only one index. The *query optimizer* (page 32) selects the index empirically by occasionally running alternate query plans and by selecting the plan with the best response time for each query type. You can override the query optimizer using the `cursor.hint()` method.
- An index "covers" a query if:
  - all the fields in the *query* (page 26) are part of that index, **and**

  - all the fields returned in the documents that match the query are in the same index.

  When an index covers a query, the server can both match the *query conditions* (page 26) **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query. Querying the index can be faster than querying the documents outside of the index.

  See *Create Indexes that Support Covered Queries* (page 200) for more information.

- Using queries with good index coverage reduces the number of full documents that MongoDB needs to store in memory, thus maximizing database performance and throughput.
- If an update does not change the size of a document or cause the document to outgrow its allocated area, then MongoDB will update an index *only if* the indexed fields have changed. This improves performance. Note that if the document has grown and must move, all index keys must then update.

6.1.2 Index Types

This section enumerates the types of indexes available in MongoDB. For all collections, MongoDB creates the default *_id index* (page 186). You can create additional indexes with the `ensureIndex()` method on any single field or *sequence of fields* (page 187) within any document or *sub-document* (page 187). MongoDB also supports indexes of arrays, called *multi-key indexes* (page 189).

_id Index

The `_id` index is a *unique index* (page 190) [1] on the `_id` field, and MongoDB creates this index by default on all collections. [2] You cannot delete the index on `_id`.

The `_id` field is the *primary key* for the collection, and every document *must* have a unique `_id` field. You may store any unique value in the `_id` field. The default value of `_id` is *ObjectId* on every insert() <db.collection.insert()‘ operation. An *ObjectId* is a 12-byte unique identifiers suitable for use as the value of an `_id` field.

Note: In *sharded clusters*, if you do *not* use the `_id` field as the *shard key*, then your application **must** ensure the uniqueness of the values in the `_id` field to prevent errors. This is most-often done by using a standard auto-generated *ObjectId*.

Secondary Indexes

All indexes in MongoDB are *secondary indexes*. You can create indexes on any field within any document or sub-document. Additionally, you can create compound indexes with multiple fields, so that a single query can match multiple components using the index while scanning fewer whole documents.

In general, you should create indexes that support your primary, common, and user-facing queries. Doing so requires MongoDB to scan the fewest number of documents possible.

In the `mongo` shell, you can create an index by calling the `ensureIndex()` method. Arguments to `ensureIndex()` resemble the following:

```
{ "field": 1 }
{ "product.quantity": 1 }
{ "product": 1, "quantity": 1 }
```

[1] Although the index on `_id` *is* unique, the `getIndexes()` method will *not* print `unique: true` in the `mongo` shell.

[2] Before version 2.2 capped collections did not have an `_id` field. In 2.2, all capped collections have an `_id` field, except those in the `local` *database*. See the *release notes* (page 399) for more information.

For each field in the index specify either `1` for an ascending order or `-1` for a descending order, which represents the order of the keys in the index. For indexes with more than one key (i.e. *compound indexes* (page 187)) the sequence of fields is important.

Indexes on Sub-documents

You can create indexes on fields that hold sub-documents as in the following example:

Example

Given the following document in the `factories` collection:

```
{ "_id": ObjectId(...), metro: { city: "New York", state: "NY" } } )
```

You can create an index on the `metro` key. The following queries would then use that index, and both would return the above document:

```
db.factories.find( { metro: { city: "New York", state: "NY" } } );
```

```
db.factories.find( { metro: { $gte : { city: "New York" } } } );
```

The second query returns the document because `{ city: "New York" }` is less than `{ city: "New York", state: "NY" }` The order of comparison is in ascending key order in the order the keys occur in the *BSON* document.

Indexes on Embedded Fields

You can create indexes on fields in sub-documents, just as you can index top-level fields in documents. [3] These indexes allow you to use a "dot notation," to introspect into sub-documents.

Consider a collection named `people` that holds documents that resemble the following example document:

```
{"_id": ObjectId(...)
 "name": "John Doe"
 "address": {
        "street": "Main"
        "zipcode": 53511
        "state": "WI"
        }
}
```

You can create an index on the `address.zipcode` field, using the following specification:

```
db.people.ensureIndex( { "address.zipcode": 1 } )
```

Compound Indexes

MongoDB supports "compound indexes," where a single index structure holds references to multiple fields within a collection's documents. Consider a collection named `products` that holds documents that resemble the following document:

[3] *Indexes on Sub-documents* (page 187), by contrast allow you to index fields that hold documents, including the full content, up to the maximum `Index Size` of the sub-document in the index.

```
{
 "_id": ObjectId(...)
 "item": "Banana"
 "category": ["food", "produce", "grocery"]
 "location": "4th Street Store"
 "stock": 4
 "type": cases
 "arrival": Date(...)
}
```

If most applications queries include the `item` field and a significant number of queries will also check the `stock` field, you can specify a single compound index to support both of these queries:

```
db.products.ensureIndex( { "item": 1, "location": 1, "stock": 1 } )
```

Compound indexes support queries on any prefix of the fields in the index. [4] For example, MongoDB can use the above index to support queries that select the `item` field and to support queries that select the `item` field **and** the `location` field. The index, however, would not support queries that select the following:

- only the `location` field
- only the `stock` field
- only the `location` and `stock` fields
- only the `item` and `stock` fields

When creating an index, the number associated with a key specifies the direction of the index. The options are `1` (ascending) and `-1` (descending). Direction doesn't matter for single key indexes or for random access retrieval but is important if you are doing sort queries on compound indexes.

The order of fields in a compound index is very important. In the previous example, the index will contain references to documents sorted first by the values of the `item` field and, within each value of the `item` field, sorted by the values of `location`, and then sorted by values of the `stock` field.

Indexes with Ascending and Descending Keys

Indexes store references to fields in either ascending or descending order. For single-field indexes, the order of keys doesn't matter, because MongoDB can traverse the index in either direction. However, for *compound indexes* (page 187), if you need to order results against two fields, sometimes you need the index fields running in opposite order relative to each other.

To specify an index with a descending order, use the following form:

```
db.products.ensureIndex( { "field": -1 } )
```

More typically in the context of a *compound index* (page 187), the specification would resemble the following prototype:

```
db.products.ensureIndex( { "fieldA": 1, "fieldB": -1 } )
```

Consider a collection of event data that includes both usernames and a timestamp. If you want to return a list of events sorted by username and then with the most recent events first. To create this index, use the following command:

```
db.events.ensureIndex( { "username" : 1, "timestamp" : -1 } )
```

[4] Index prefixes are the beginning subset of fields. For example, given the index `{ a: 1, b: 1, c: 1 }` both `{ a: 1 }` and `{ a: 1, b: 1 }` are prefixes of the index.

Multikey Indexes

If you index a field that contains an array, MongoDB indexes each value in the array separately, in a "multikey index."

Example

Given the following document:

```
{ "_id" : ObjectId("..."),
  "name" : "Warm Weather",
  "author" : "Steve",
  "tags" : [ "weather", "hot", "record", "april" ] }
```

Then an index on the `tags` field would be a multikey index and would include these separate entries:

```
{ tags: "weather" }
{ tags: "hot" }
{ tags: "record" }
{ tags: "april" }
```

Queries could use the multikey index to return queries for any of the above values.

You can use multikey indexes to index fields within objects embedded in arrays, as in the following example:

Example

Consider a `feedback` collection with documents in the following form:

```
{
 "_id": ObjectId(...)
 "title": "Grocery Quality"
 "comments": [
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the cheddar selection." },
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the mustard selection." },
    { author_id: ObjectId(...)
      date: Date(...)
      text: "Please expand the olive selection." }
 ]
}
```

An index on the `comments.text` field would be a multikey index and would add items to the index for all of the sub-documents in the array.

With an index, such as `{ comments.text: 1 }`, consider the following query:

```
db.feedback.find( { "comments.text": "Please expand the olive selection." } )
```

This would select the document, that contains the following document in the `comments.text` array:

```
{ author_id: ObjectId(...)
  date: Date(...)
  text: "Please expand the olive selection." }
```

Compound Multikey Indexes May Only Include One Array Field

While you can create multikey *compound indexes* (page 187), at most one field in a compound index may hold an array. For example, given an index on `{ a: 1, b: 1 }`, the following documents are permissible:

```
{a: [1, 2], b: 1}
```

```
{a: 1, b: [1, 2]}
```

However, the following document is impermissible, and MongoDB cannot insert such a document into a collection with the `{a: 1, b: 1 }` index:

```
{a: [1, 2], b: [1, 2]}
```

If you attempt to insert a such a document, MongoDB will reject the insertion, and produce an error that says `cannot index parallel arrays`. MongoDB does not index parallel arrays because they require the index to include each value in the Cartesian product of the compound keys, which could quickly result in incredibly large and difficult to maintain indexes.

Unique Indexes

A unique index causes MongoDB to reject all documents that contain a duplicate value for the indexed field. To create a unique index on the `user_id` field of the `members` collection, use the following operation in the `mongo` shell:

```
db.addresses.ensureIndex( { "user_id": 1 }, { unique: true } )
```

By default, `unique` is `false` on MongoDB indexes.

If you use the unique constraint on a *compound index* (page 187) then MongoDB will enforce uniqueness on the *combination* of values, rather than the individual value for any or all values of the key.

If a document does not have a value for the indexed field in a unique index, the index will store a null value for this document. MongoDB will only permit one document without a unique value in the collection because of this unique constraint. You can combine with the *sparse index* (page 190) to filter these null values from the unique index.

Sparse Indexes

Sparse indexes only contain entries for documents that have the indexed field. [5] Any document that is missing the field is not indexed. The index is "sparse" because of the missing documents when values are missing.

By contrast, non-sparse indexes contain all documents in a collection, and store null values for documents that do not contain the indexed field. Create a sparse index on the `xmpp_id` field, of the `members` collection, using the following operation in the `mongo` shell:

```
db.addresses.ensureIndex( { "xmpp_id": 1 }, { sparse: true } )
```

By default, `sparse` is `false` on MongoDB indexes.

Warning: Using these indexes will sometimes result in incomplete results when filtering or sorting results, because sparse indexes are not complete for all documents in a collection.

Note: Do not confuse sparse indexes in MongoDB with block-level[6] indexes in other databases. Think of them as dense indexes with a specific filter.

You can combine the sparse index option with the *unique indexes* (page 190) option so that `mongod` will reject documents that have duplicate values for a field, but that ignore documents that do not have the key.

[5] All documents that have the indexed field *are* indexed in a sparse index, even if that field stores a null value in some documents.
[6] http://en.wikipedia.org/wiki/Database_index#Sparse_index

6.1.3 Index Creation Options

You specify index creation options in the second argument in `ensureIndex()`.

The options *sparse* (page 190), *unique* (page 190), and *TTL* (page 192) affect the kind of index that MongoDB creates. This section addresses, *background construction* (page 191) and *duplicate dropping* (page 192), which affect how MongoDB builds the indexes.

Background Construction

By default, creating an index is a blocking operation. Building an index on a large collection of data can take a long time to complete. To resolve this issue, the background option can allow you to continue to use your `mongod` instance during the index build.

For example, to create an index in the background of the `zipcode` field of the `people` collection you would issue the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true} )
```

By default, `background` is `false` for building MongoDB indexes.

You can combine the background option with other options, as in the following:

```
db.people.ensureIndex( { zipcode: 1}, {background: true, sparse: true } )
```

Be aware of the following behaviors with background index construction:

- A `mongod` instance can only build one background index per database, at a time.

  Changed in version 2.2: Before 2.2, a single `mongod` instance could only build one index at a time.

- The indexing operation runs in the background so that other database operations can run while creating the index. However, the `mongo` shell session or connection where you are creating the index will block until the index build is complete. Open another connection or `mongo` instance to continue using commands to the database.

- The background index operation use an incremental approach that is slower than the normal "foreground" index builds. If the index is larger than the available RAM, then the incremental process can take *much* longer than the foreground build.

- If your application includes `ensureIndex()` operations, and an index *doesn't* exist for other operational concerns, building the index can have a severe impact on the performance of the database.

  Make sure that your application checks for the indexes at start up using the `getIndexes()` method or the equivalent method for your driver[7] and terminates if the proper indexes do not exist. Always build indexes in production instances using separate application code, during designated maintenance windows.

Building Indexes on Secondaries

Background index operations on a *replica set primary* become foreground indexing operations on secondary members of the set. All indexing operations on secondaries block replication.

To build large indexes on secondaries the best approach is to restart one secondary at a time in *standalone* mode and build the index. After building the index, restart as a member of the replica set, allow it to catch up with the other members of the set, and then build the index on the next secondary. When all the secondaries have the new index, step down the primary, restart it as a standalone, and build the index on the former primary.

[7] http://api.mongodb.org/

Remember, the amount of time required to build the index on a secondary node must be within the window of the *oplog*, so that the secondary can catch up with the primary.

See *Build Indexes on Replica Sets* (page 198) for more information on this process.

Indexes on secondary members in "recovering" mode are always built in the foreground to allow them to catch up as soon as possible.

See *Build Indexes on Replica Sets* (page 198) for a complete procedure for rebuilding indexes on secondaries.

Note: If MongoDB is building an index in the background, you cannot perform other administrative operations involving that collection, including `repairDatabase`, drop that collection (i.e. `db.collection.drop()`,) and `compact`. These operations will return an error during background index builds.

Queries will not use these indexes until the index build is complete.

Drop Duplicates

MongoDB cannot create a *unique index* (page 190) on a field that has duplicate values. To force the creation of a unique index, you can specify the `dropDups` option, which will only index the first occurrence of a value for the key, and delete all subsequent values.

Warning: As in all unique indexes, if a document does not have the indexed field, MongoDB will include it in the index with a "null" value.
If subsequent fields *do not* have the indexed field, and you have set `{dropDups: true}`, MongoDB will remove these documents from the collection when creating the index. If you combine `dropDups` with the *sparse* (page 190) option, this index will only include documents in the index that have the value, and the documents without the field will remain in the database.

To create a unique index that drops duplicates on the `username` field of the `accounts` collection, use a command in the following form:

```
db.accounts.ensureIndex( { username: 1 }, { unique: true, dropDups: true } )
```

Warning: Specifying `{ dropDups: true }` will delete data from your database. Use with extreme caution.

By default, `dropDups` is `false`.

6.1.4 Index Features

TTL Indexes

TTL indexes are special indexes that MongoDB can use to automatically remove documents from a collection after a certain amount of time. This is ideal for some types of information like machine generated event data, logs, and session information that only need to persist in a database for a limited amount of time.

These indexes have the following limitations:

- *Compound indexes* (page 187) are *not* supported.
- The indexed field **must** be a date *type*.
- If the field holds an array, and there are multiple date-typed data in the index, the document will expire when the *lowest* (i.e. earliest) matches the expiration threshold.

Note: TTL indexes expire data by removing documents in a background task that runs *every 60 seconds*. As a result, the TTL index provides no guarantees that expired documents will not exist in the collection. Consider that:

- Documents may remain in a collection *after* they expire and before the background process runs.
- The duration of the removal operations depend on the workload of your `mongod` instance.

In all other respects, TTL indexes are normal indexes, and if appropriate, MongoDB can use these indexes to fulfill arbitrary queries.

See

`http://docs.mongodb.org/manual/tutorial/expire-data`

Geospatial Indexes

MongoDB provides "geospatial indexes" to support location-based and other similar queries in a two dimensional coordinate systems. For example, use geospatial indexes when you need to take a collection of documents that have coordinates, and return a number of options that are "near" a given coordinate pair.

To create a geospatial index, your *documents* must have a coordinate pair. For maximum compatibility, these coordinate pairs should be in the form of a two element array, such as `[ x , y ]`. Given the field of `loc`, that held a coordinate pair, in the collection `places`, you would create a geospatial index as follows:

```
db.places.ensureIndex( { loc : "2d" } )
```

MongoDB will reject documents that have values in the `loc` field beyond the minimum and maximum values.

Note: MongoDB permits only one geospatial index per collection. Although, MongoDB will allow clients to create multiple geospatial indexes, a single query can use only one index.

See the `$near`, and the database command `geoNear` for more information on accessing geospatial data.

Geohaystack Indexes

In addition to conventional *geospatial indexes* (page 193), MongoDB also provides a bucket-based geospatial index, called "geospatial haystack indexes." These indexes support high performance queries for locations within a small area, when the query must filter along another dimension.

Example

If you need to return all documents that have coordinates within 25 miles of a given point *and* have a type field value of "museum," a haystack index would be provide the best support for these queries.

Haystack indexes allow you to tune your bucket size to the distribution of your data, so that in general you search only very small regions of 2d space for a particular kind of document. These indexes are not suited for finding the closest documents to a particular location, when the closest documents are far away compared to bucket size.

6.1.5 Index Behaviors and Limitations

Be aware of the following behaviors and limitations:

- A collection may have no more than *64 indexes*.

- Index keys can be no larger than *1024 bytes*.

  Documents with fields that have values greater than this size cannot be indexed.

  To query for documents that were too large to index, you can use a command similar to the following:

  ```
  db.myCollection.find({<key>: <value too large to index>}).hint({$natural: 1})
  ```

- The name of an index, including the *namespace* must be shorter than *128 characters*.
- Indexes have storage requirements, and impacts insert/update speed to some degree.
- Create indexes to support queries and other operations, but do not maintain indexes that your MongoDB instance cannot or will not use.
- For queries with the `$or` operator, each clause of an `$or` query executes in parallel, and can each use a different index.
- For queries that use the `sort()` method and use the `$or` operator, the query **cannot** use the indexes on the `$or` fields.
- `2d` *geospatial queries* (page 209) do not support queries that use the `$or` operator.

6.2 Indexing Operations

This document provides operational guidelines and procedures for indexing data in MongoDB collections. For the fundamentals of MongoDB indexing, see the *Indexing Overview* (page 185) document. For strategies and practical approaches, see the *Indexing Strategies* (page 199) document.

Indexes allow MongoDB to process and fulfill queries quickly by creating small and efficient representations of the documents in a collection.

6.2.1 Create an Index

To create an index, use `db.collection.ensureIndex()` or a similar method from your driver[8]. For example the following creates an index on the `phone-number` field of the `people` collection:

```
db.people.ensureIndex( { "phone-number": 1 } )
```

`ensureIndex()` only creates an index if an index of the same specification does not already exist.

All indexes support and optimize the performance for queries that select on this field. For queries that cannot use an index, MongoDB must scan all documents in a collection for documents that match the query.

Example

If you create an index on the `user_id` field in the `records`, this index is, the index will support the following query:

```
db.records.find( { user_id: 2 } )
```

However, the following query, on the `profile_url` field is not supported by this index:

```
db.records.find( { profile_url: 2 } )
```

[8] http://api.mongodb.org/

If your collection holds a large amount of data, consider building the index in the background, as described in *Background Construction* (page 191). To build indexes on replica sets, see the *Build Indexes on Replica Sets* (page 198) section for more information.

6.2.2 Create a Compound Index

To create a *compound index* (page 187) use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1, c: 1 } )
```

For example, the following operation will create an index on the `item`, `category`, and `price` fields of the `products` collection:

```
db.products.ensureIndex( { item: 1, category: 1, price: 1 } )
```

Some drivers may specify indexes, using `NumberLong(1)` rather than `1` as the specification. This does not have any affect on the resulting index.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 198).

If your collection is large, build the index in the background, as described in *Background Construction* (page 191). If you build in the background on a live replica set, see also *Build Indexes on Replica Sets* (page 198).

6.2.3 Special Creation Options

Note: TTL collections use a special `expire` index option. See `http://docs.mongodb.org/manual/tutorial/expire-data` for more information.

Note: To create geospatial indexes, see *Create a Geospatial Index* (page 210).

Sparse Indexes

To create a *sparse index* (page 190) on a field, use an operation that resembles the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { sparse: true } )
```

The following example creates a sparse index on the `users` table that *only* indexes the `twitter_name` *if* a document has this field. This index will not include documents in this collection without the `twitter_name` field.

```
db.users.ensureIndex( { twitter_name: 1 }, { sparse: true } )
```

Note: Sparse indexes can affect the results returned by the query, particularly with respect to sorts on fields *not* included in the index. See the *sparse index* (page 190) section for more information.

Unique Indexes

To create a *unique indexes* (page 190), consider the following prototype:

```
db.collection.ensureIndex( { a: 1 }, { unique: true } )
```

For example, you may want to create a unique index on the `"tax-id":` of the `accounts` collection to prevent storing multiple account records for the same legal entity:

```
db.accounts.ensureIndex( { "tax-id": 1 }, { unique: true } )
```

The *_id index* (page 186) is a unique index. In some situations you may consider using `_id` field itself for this kind of data rather than using a unique index on another field.

In many situations you will want to combine the `unique` constraint with the `sparse` option. When MongoDB indexes a field, if a document does not have a value for a field, the index entry for that item will be `null`. Since unique indexes cannot have duplicate values for a field, without the `sparse` option, MongoDB will reject the second document and all subsequent documents without the indexed field. Consider the following prototype.

```
db.collection.ensureIndex( { a: 1 }, { unique: true, sparse: true } )
```

You can also enforce a unique constraint on *compound indexes* (page 187), as in the following prototype:

```
db.collection.ensureIndex( { a: 1, b: 1 }, { unique: true } )
```

These indexes enforce uniqueness for the *combination* of index keys and *not* for either key individually.

Create in Background

To create an index in the background you can specify *background construction* (page 191). Consider the following prototype invocation of `db.collection.ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { background: true } )
```

Consider the section on *background index construction* (page 191) for more information about these indexes and their implications.

Drop Duplicates

To force the creation of a *unique index* (page 190) index on a collection with duplicate values in the field you are indexing you can use the `dropDups` option. This will force MongoDB to create a *unique* index by deleting documents with duplicate values when building the index. Consider the following prototype invocation of `db.collection.ensureIndex()`:

```
db.collection.ensureIndex( { a: 1 }, { dropDups: true } )
```

See the full documentation of *duplicate dropping* (page 192) for more information.

> **Warning:** Specifying `{ dropDups: true }` may delete data from your database. Use with extreme caution.

Refer to the `ensureIndex()` documentation for additional index creation options.

6.2.4 Information about Indexes

List all Indexes on a Collection

To return a list of all indexes on a collection, use the, use the `db.collection.getIndexes()` method or a similar method for your driver[9].

For example, to view all indexes on the `people` collection:

```
db.people.getIndexes()
```

List all Indexes for a Database

To return a list of all indexes on all collections in a database, use the following operation in the `mongo` shell:

```
db.system.indexes.find()
```

Measure Index Use

Query performance is a good general indicator of index use; however, for more precise insight into index use, MongoDB provides the following tools:

- `explain()`

  Append the `explain()` method to any cursor (e.g. query) to return a document with statistics about the query process, including the index used, the number of documents scanned, and the time the query takes to process in milliseconds.

- `cursor.hint()`

  Append the `hint()` to any cursor (e.g. query) with the index as the argument to *force* MongoDB to use a specific index to fulfill the query. Consider the following example:

  ```
  db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } } ).hint( { zipcode: 1 } )
  ```

  You can use `hint()` and `explain()` in conjunction with each other to compare the effectiveness of a specific index. Specify the `$natural` operator to the `hint()` method to prevent MongoDB from using *any* index:

  ```
  db.people.find( { name: "John Doe", zipcode: { $gt: 63000 } } } ).hint( { $natural: 1 } )
  ```

- `indexCounters`

  Use the `indexCounters` data in the output of `serverStatus` for insight into database-wise index utilization.

6.2.5 Remove Indexes

To remove an index, use the `db.collection.dropIndex()` method, as in the following example:

```
db.accounts.dropIndex( { "tax-id": 1 } )
```

This will remove the index on the `"tax-id"` field in the `accounts` collection. The shell provides the following document after completing the operation:

[9] http://api.mongodb.org/

```
{ "nIndexesWas" : 3, "ok" : 1 }
```

Where the value of `nIndexesWas` reflects the number of indexes *before* removing this index. You can also use the `db.collection.dropIndexes()` to remove *all* indexes, except for the *_id index* (page 186) 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.

6.2.6 Rebuild Indexes

If you need to rebuild indexes for a collection you can use the `db.collection.reIndex()` method. This will drop all indexes, including the *_id index* (page 186), and then rebuild all indexes. The operation takes the following form:

```
db.accounts.reIndex()
```

MongoDB will return the following document when the operation completes:

```
{
        "nIndexesWas" : 2,
        "msg" : "indexes dropped for collection",
        "nIndexes" : 2,
        "indexes" : [
                {
                        "key" : {
                                "_id" : 1,
                                "tax-id" : 1
                        },
                        "ns" : "records.accounts",
                        "name" : "_id_"
                }
        ],
        "ok" : 1
}
```

This shell helper provides a wrapper around the `reIndex` *database command*. Your `client library` may have a different or additional interface for this operation.

Note: To build or rebuild indexes for a *replica set* see *Build Indexes on Replica Sets* (page 198).

6.2.7 Build Indexes on Replica Sets

Background index creation operations (page 191) become *foreground* indexing operations on *secondary* members of replica sets. The foreground index building process blocks all replication and read operations on the secondaries while they build the index.

Secondaries will begin building indexes *after* the *primary* finishes building the index. In *sharded clusters*, the `mongos` will send `ensureIndex()` to the primary members of the replica set for each shard, which then replicate to the secondaries after the primary finishes building the index.

To minimize the impact of building an index on your replica set, use the following procedure to build indexes on secondaries:

Note: If you need to build an index in a *sharded cluster*, repeat the following procedure for each replica set that provides each *shard*.

1. Stop the `mongod` process on one secondary. Restart the `mongod` process *without* the `--replSet` option and running on a different port. [10] This instance is now in "standalone" mode.
2. Create the new index or rebuild the index on this `mongod` instance.
3. Restart the `mongod` instance with the `--replSet` option. Allow replication to catch up on this member.
4. Repeat this operation on all of the remaining secondaries.
5. Run `rs.stepDown()` (page 288) on the *primary* member of the set, and then repeat this procedure on the former primary.

Warning: Ensure that your *oplog* is large enough to permit the indexing or re-indexing operation to complete without falling too far behind to catch up. See the "*oplog sizing* (page 220)" documentation for additional information.

Note: This procedure *does* take one member out of the replica set at a time. However, this procedure will only affect one member of the set at a time rather than *all* secondaries at the same time.

6.2.8 Monitor and Control Index Building

To see the status of the indexing processes, you can use the `db.currentOp()` method in the `mongo` shell. The value of the `query` field and the `msg` field will indicate if the operation is an index build. The `msg` field also indicates the percent of the build that is complete.

You can only terminate a background index build. If you need to terminate an ongoing index build, You can use the `db.killOp()` method in the `mongo` shell.

6.3 Indexing Strategies

This document provides strategies for indexing in MongoDB. For fundamentals of MongoDB indexing, see *Indexing Overview* (page 185). For operational guidelines and procedures, see *Indexing Operations* (page 194).

6.3.1 Strategies

The best indexes for your application are based on a number of factors, including the kinds of queries you expect, the ratio of reads to writes, and the amount of free memory on your system.

When developing your indexing strategy you should have a deep understanding of:

- The application's queries.
- The relative frequency of each query in the application.
- The current indexes created for your collections.
- Which indexes the most common queries use.

The best overall strategy for designing indexes is to profile a variety of index configurations with data sets similar to the ones you'll be running in production to see which configurations perform best.

MongoDB can only use *one* index to support any given operation. However, each clause of an `$or` query may use a different index.

[10] 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.

6.3.2 Create Indexes to Support Your Queries

If you only ever query on a single key in a given collection, then you need to create just one single-key index for that collection. For example, you might create an index on `category` in the `product` collection:

```
db.products.ensureIndex( { "category": 1 } )
```

However, 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* (page 187) is more efficient. MongoDB will use the compound index for both queries. For example, you might create an index on both `category` and `item`.

```
db.products.ensureIndex( { "category": 1, "item": 1 } )
```

This allows you both options. You can query on just `category`, and you also can query on `category` combined with `item`. (To query on multiple keys and sort the results, see *Use Indexes to Sort Query Results* (page 201).)

With the exception of queries that use the `$or` operator, a query does not use multiple indexes. A query uses only one index.

6.3.3 Use Compound Indexes to Support Several Different Queries

A single *compound index* (page 187) on multiple fields can support all the queries that search a "prefix" subset of those fields.

Example

The following index on a collection:

```
{ x: 1, y: 1, z: 1 }
```

Can support queries that the following indexes support:

```
{ x: 1 }
{ x: 1, y: 1 }
```

There are some situations where the prefix indexes may offer better query performance: for example if `z` is a large array.

The `{ x: 1, y: 1, z: 1 }` index can also support many of the same queries as the following index:

```
{ x: 1, z: 1 }
```

Also, `{ x: 1, z: 1 }` has an additional use. Given the following query:

```
db.collection.find( { x: 5 } ).sort( { z: 1} )
```

The `{ x: 1, z: 1 }` index supports both the query and the sort operation, while the `{ x: 1, y: 1, z: 1 }` index only supports the query. For more information on sorting, see *Use Indexes to Sort Query Results* (page 201).

6.3.4 Create Indexes that Support Covered Queries

A covered query is a query in which:

- all the fields in the *query* (page 26) are part of an index, **and**
- all the fields returned in the results are in the same index.

Because the index "covers" the query, MongoDB can both match the *query conditions* (page 26) **and** return the results using only the index; MongoDB does not need to look at the documents, only the index, to fulfill the query.

Querying *only* the index can be much faster than querying documents outside of the index. Index keys are typically smaller than the documents they catalog, and indexes are typically available in RAM or located sequentially on disk.

MongoDB automatically uses an index that covers a query when possible. To ensure that an index can *cover* a query, create an index that includes all the fields listed in the *query document* (page 26) and in the query result. You can specify the fields to return in the query results with a *projection* (page 29) document. By default, MongoDB includes the `_id` field in the query result. So, if the index does **not** include the `_id` field, then you must exclude the `_id` field (i.e. `_id: 0`) from the query results.

Consider the following example where the collection `user` has an index on the fields `user` and `status`:

```
{ status: 1, user: 1 }
```

Then, the following query which queries on the `status` field and returns only the `user` field is covered:

```
db.users.find( { status: "A" }, { user: 1, _id: 0 } )
```

However, the following query that uses the index to match documents is **not** covered by the index because it returns both the `user` field **and** the `_id` field:

```
db.users.find( { status: "A" }, { user: 1 } )
```

An index **cannot** cover a query if:

- any of the indexed fields in any of the documents in the collection includes an array. If an indexed field is an array, the index becomes a *multi-key index* (page 189) index and cannot support a covered query.
- any of the indexed fields are fields in subdocuments. To index fields in subdocuments, use *dot notation*. For example, consider a collection `users` with documents of the following form:

  ```
  { _id: 1, user: { login: "tester" } }
  ```

  The collection has the following indexes:

  ```
  { user: 1 }
  ```

  ```
  { "user.login": 1 }
  ```

  The `{ user: 1 }` index covers the following query:

  ```
  db.users.find( { user: { login: "tester" } }, { user: 1, _id: 0 } )
  ```

  However, the `{ "user.login": 1 }` index does **not** cover the following query:

  ```
  db.users.find( { "user.login": "tester" }, { "user.login": 1, _id: 0 } )
  ```

  The query, however, does use the `{ "user.login": 1 }` index to find matching documents.

To determine whether a query is a covered query, use the `explain()` method. If the `explain()` output displays `true` for the `indexOnly` field, the query is covered by an index, and MongoDB queries only that index to match the query **and** return the results.

For more information see *Measure Index Use* (page 197).

6.3.5 Use Indexes to Sort Query Results

For the fastest performance when sorting query results by a given field, create a sorted index on that field.

To sort query results on multiple fields, create a *compound index* (page 187). MongoDB sorts results based on the field order in the index. For queries that include a sort that uses a compound index, ensure that all fields before the first sorted field are equality matches.

Example

If you create the following index:

```
{ a: 1, b: 1, c: 1, d: 1 }
```

The following query and sort operations can use the index:

```
db.collection.find().sort( { a:1 } )
db.collection.find().sort( { a:1, b:1 } )

db.collection.find( { a:4 } ).sort( { a:1, b:1 } )
db.collection.find( { b:5 } ).sort( { a:1, b:1 } )

db.collection.find( { a:5 } ).sort( { b:1, c:1 } )

db.collection.find( { a:5, c:4, b:3 } ).sort( { d:1 } )

db.collection.find( { a: { $gt:4 } } ).sort( { a:1, b:1 } )
db.collection.find( { a: { $gt:5 } } ).sort( { a:1, b:1 } )

db.collection.find( { a:5, b:3, d:{ $gt:4 } } ).sort( { c:1 } )
db.collection.find( { a:5, b:3, c:{ $lt:2 }, d:{ $gt:4 } } ).sort( { c:1 } )
```

However, the following queries cannot sort the results using the index:

```
db.collection.find().sort( { b:1 } )
db.collection.find( { b:5 } ).sort( { b:1 } )
```

Note: For in-memory sorts that do not use an index, the `sort()` operation is significantly slower. The `sort()` operation will abort when it uses 32 megabytes of memory.

6.3.6 Ensure Indexes Fit RAM

For the fastest processing, ensure that your indexes fit entirely in RAM so that the system can avoid reading the index from disk.

To check the size of your indexes, use the `db.collection.totalIndexSize()` helper, which returns data in bytes:

```
> db.collection.totalIndexSize()
4294976499
```

The above example shows an index size of almost 4.3 gigabytes. To ensure this index fits in RAM, you must not only have more than that much RAM available but also must have RAM available for the rest of the *working set*. Also remember:

If you have and use multiple collections, you must consider the size of all indexes on all collections. The indexes and the working set must be able to fit in memory at the same time.

There are some limited cases where indexes do not need to fit in memory. See *Indexes that Hold Only Recent Values in RAM* (page 203).

See also:

For additional `collection statistics`, use `collStats` or `db.collection.stats()`.

Indexes that Hold Only Recent Values in RAM

Indexes do not have to fit *entirely* into RAM in all cases. If the value of the indexed field 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.

6.3.7 Create Queries that Ensure Selectivity

Selectivity is the ability of a query to narrow results using the index. Effective indexes are more selective and allow MongoDB to use the index for a larger portion of the work associated with fulfilling the query.

To ensure selectivity, write queries that limit the number of possible documents with the indexed field. Write queries that are appropriately selective relative to your indexed data.

Example

Suppose you have a field called `status` where the possible values are `new` and `processed`. If you add an index on `status` you've created a low-selectivity index. The index will be of little help in locating records.

A better strategy, depending on your queries, would be to create a *compound index* (page 187) that includes the low-selectivity field and another field. For example, you could create a compound index on `status` and `created_at`.

Another option, again depending on your use case, might be to use separate collections, one for each status.

Example

Consider an index `{ a : 1 }` (i.e. an index on the key `a` sorted in ascending order) on a collection where `a` has three values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 1, b: "cd" }
{ _id: ObjectId(), a: 1, b: "ef" }
{ _id: ObjectId(), a: 2, b: "jk" }
{ _id: ObjectId(), a: 2, b: "lm" }
{ _id: ObjectId(), a: 2, b: "no" }
{ _id: ObjectId(), a: 3, b: "pq" }
{ _id: ObjectId(), a: 3, b: "rs" }
{ _id: ObjectId(), a: 3, b: "tv" }
```

If you query for `{ a: 2, b: "no" }` MongoDB must scan 3 *documents* in the collection to return the one matching result. Similarly, a query for `{ a: { $gt: 1}, b: "tv" }` must scan 6 documents, also to return one result.

Consider the same index on a collection where `a` has *nine* values evenly distributed across the collection:

```
{ _id: ObjectId(), a: 1, b: "ab" }
{ _id: ObjectId(), a: 2, b: "cd" }
{ _id: ObjectId(), a: 3, b: "ef" }
{ _id: ObjectId(), a: 4, b: "jk" }
{ _id: ObjectId(), a: 5, b: "lm" }
{ _id: ObjectId(), a: 6, b: "no" }
{ _id: ObjectId(), a: 7, b: "pq" }
{ _id: ObjectId(), a: 8, b: "rs" }
{ _id: ObjectId(), a: 9, b: "tv" }
```

If you query for `{ a: 2, b: "cd" }`, MongoDB must scan only one document to fulfill the query. The index and query are more selective because the values of `a` are evenly distributed *and* the query can select a specific document using the index.

However, although the index on `a` is more selective, a query such as `{ a: { $gt: 5 }, b: "tv" }` would still need to scan 4 documents.

If overall selectivity is low, and if MongoDB must read a number of documents to return results, then some queries may perform faster without indexes. To determine performance, see *Measure Index Use* (page 197).

6.3.8 Consider Performance when Creating Indexes for Write-heavy Applications

If your application is write-heavy, then be careful when creating new indexes, since each additional index with impose a write-performance penalty. In general, don't be careless about adding indexes. Add indexes to complement your queries. Always have a good reason for adding a new index, and be sure to benchmark alternative strategies.

Consider Insert Throughput

MongoDB must update *all* indexes associated with a collection after every insert, update, or delete operation. For update operations, if the updated document does not move to a new location, then MongoDB only modifies the updated fields in the index. Therefore, every index on a collection adds some amount of overhead to these write operations. In almost every case, the performance gains that indexes realize for read operations are worth the insertion penalty. However, in some cases:

- An index to support an infrequent query might incur more insert-related costs than savings in read-time.
- If you have many indexes on a collection with a high insert throughput and a number of related indexes, you may find better overall performance with a smaller number of indexes, even if some queries are less optimally supported by an index.
- If your indexes and queries are not sufficiently *selective* (page 203), the speed improvements for query operations may not offset the costs of maintaining an index. For more information see *Create Queries that Ensure Selectivity* (page 203).

6.4 Geospatial Queries with `2d` Indexes

MongoDB provides support for querying location-based data using special geospatial indexes. For an introduction to these `2d` indexes, see *2d Geospatial Indexes* (page 209).

MongoDB supports the following geospatial query types:

- Proximity queries, which select documents based on distance to a given point. See *Proximity Queries* (page 205).
- Bounded queries, which select documents that have coordinates within a specified area. See *Bounded Queries* (page 206).
- Exact queries, which select documents with an exact coordinate pair, which has limited applicability. See *Query for Exact Matches* (page 207).

6.4.1 Considerations

With the `geoNear` command, a collection can have only one `2d` index. With *geospatial query operators* such as `$near` operator, a collection can have multiple geospatial indexes.

6.4.2 Proximity Queries

Proximity queries select the documents closest to the point specified in the query. To perform proximity queries you use either the `find()` method with the `$near` operator or you use the `geoNear` command.

The `find()` method with the `$near` operator returns 100 documents by default and sorts the results by distance. The `$near` operator uses the following form:

```
db.collection.find( { <location field>: { $near: [ x, y ] } } )
```

Example

The following query

```
db.places.find( { loc: { $near: [ -70, 40 ] } } )
```

returns output similar to the following:

```
{ "_id" : ObjectId(" ... "), "loc" : [ -73, 39 ] }
```

The `geoNear` command returns more information than does the `$near` operator. The `geoNear` command can only return a single 16-megabyte result set. The `geoNear` command also offers additional operators, such as operators to query for *maximum* (page 206) or *spherical* (page 208) distance. For a list of operators, see `geoNear`.

Without additional operators, the `geoNear` command uses the following form:

```
db.runCommand( { geoNear: <collection>, near: [ x, y ] } )
```

Example

The following command returns the same results as the `$near` in the previous example but with more information:

```
db.runCommand( { geoNear: "places", near: [ -74, 40.74 ] } )
```

This operation will return the following output:

```
{
   "ns" : "test.places",
   "results" : [
      {
         "dis" : 3,
         "obj" : {
            "_id" : ObjectId(" ... "),
            "loc" : [
               -73,
               39
            ]
         }
      }
   ],
   "stats" : {
      "time" : 2,
      "btreelocs" : 0,
      "nscanned" : 1,
      "objectsLoaded" : 1,
      "avgDistance" : 3,
      "maxDistance" : 3.0000188685220253
   },
   "near" : "0110000111111000000111111000000111111000000111111000",
```

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

6.4.3 Distance Queries

You can limit a proximity query to those documents that fall within a maximum distance of a point. You specify the maximum distance using the units specified by the coordinate system. For example, if the coordinate system uses meters, you specify maximum distance in meters.

To specify distance using the `find()` method, use `$maxDistance` operator. Use the following form:

```
db.collection.find( { <location field> : { $near : [ x , y ] , $maxDistance : <distance> } } )
```

To specify distance with the `geoNear` command, use the `maxDistance` option. Use the following form:

```
db.runCommand( { geoNear: <collection>, near: [ x, y ], maxDistance: <distance> } )
```

6.4.4 Limit the Number of Results

By default, geospatial queries using `find()` method return 100 documents, sorted by distance. To limit the result when using the `find()` method, use the `limit()` method, as in the following prototype:

```
db.collection.find( { <location field>: { $near: [ x, y ] } } ).limit(<n>)
```

To limit the result set when using the `geoNear` command, use the `num` option. Use the following form:

```
db.runCommand( { geoNear: <collection>, near: [ x, y ], num: z } )
```

To limit geospatial search results by distance, see *Distance Queries* (page 206).

Bounded Queries

Bounded queries return documents within a shape defined using the `$within` operator. MongoDB's bounded queries support the following shapes:

- *Circles* (page 207)
- *Rectangles* (page 207)
- *Polygons* (page 207)

Bounded queries do not return sorted results. As a result MongoDB can return bounded queries more quickly than *proximity queries* (page 205). Bounded queries have the following form:

```
db.collection.find( { <location field> :
                          { "$within" :
                            { <shape> : <shape dimensions> }
                          }
                    } )
```

The following sections describe each of the shapes supported by bounded queries:

6.4.5 Circles

To query for documents with coordinates inside the bounds of a circle, specify the center and the radius of the circle using the `$within` operator and `$center` option. Consider the following prototype query:

```
db.collection.find( { "field": { "$within": { "$center": [ center, radius ] } } } )
```

The following example query returns all documents that have coordinates that exist within the circle centered on `[-74, 40.74]` and with a radius of `10`, using a geospatial index on the `loc` field:

```
db.places.find( { "loc": { "$within":
                           { "$center": [ [-74, 40.74], 10 ] }
                         }
                } )
```

The `$within` operator using `$center` is similar to using `$maxDistance`, but `$center` has different performance characteristics. MongoDB does not sort queries that use the `$within` operator are not sorted, unlike queries using the `$near` operator.

6.4.6 Rectangles

To query for documents with coordinates inside the bounds of a rectangle, specify the lower-left and upper-right corners of the rectangle using the `$within` operator and `$box` option. Consider the following prototype query:

```
db.collection.find( { "field": { "$within": { "$box": [ coordinate0, coordinate1 ] } } } )
```

The following query returns all documents that have coordinates that exist within the rectangle where the lower-left corner is at `[ 0, 0 ]` and the upper-right corner is at `[ 3, 3 ]`, using a geospatial index on the `loc` field:

```
db.places.find( { "loc": { "$within": { "$box": [ [0, 0] , [3, 3] ] } } } )
```

6.4.7 Polygons

New in version 1.9: Support for polygon queries.

To query for documents with coordinates inside of a polygon, specify the points of the polygon in an array, using the `$within` operator with the `$polygon` option. MongoDB automatically connects the last point in the array to the first point. Consider the following prototype query:

```
db.places.find({ "loc": { "$within": { "$polygon": [ points ] } } })
```

The following query returns all documents that have coordinates that exist within the polygon defined by `[ [0,0], [3,3], [6,0] ]`:

```
db.places.find({ "loc": { "$within": { "$polygon":
                                       [ [ 0,0], [3,3], [6,0] ] } } } )
```

Query for Exact Matches

You can use the `db.collection.find()` method to query for an exact match on a location. These queries have the following form:

```
db.collection.find( { <location field>: [ x, y ] } )
```

This query will return any documents with the value of `[ x, y ]`.

Exact geospatial queries only applicability for a limited selection of cases, the *proximity* (page 205) and *bounded* (page 206) queries provide more useful results for more applications.

Calculate Distances Using Spherical Geometry

When you query using the `2d` index, MongoDB calculates distances using flat geometry by default, which models points on a flat surface.

Optionally, you may instruct MongoDB to calculate distances using spherical geometry, which models points on a spherical surface. Spherical geometry is useful for modeling coordinates on the surface of Earth.

To calculate distances using spherical geometry, use MongoDB's spherical query operators and options:

- `find()` method with the `$nearSphere` operator.
- `find()` method with the `$centerSphere`.
- `geoNear` command with the `{ spherical: true }` option.

See also:

geospatial-query-operators.

For more information on differences between flat and spherical distance calculation, see *Distance Calculation* (page 212).

Distance Multiplier

The `distanceMultiplier` option `geoNear` returns distances only after multiplying the results by command by an assigned value. This allows MongoDB to return converted values, and removes the requirement to convert units in application logic.

Note: Because `distanceMultiplier` is an option to `geoNear`, the multiplication operation occurs on the `mongod` process. The operation adds a slight overhead to the operation of `geoNear`.

Using `distanceMultiplier` in spherical queries allows one to use results from the `geoNear` command without radian to distance conversion. The following example uses `distanceMultiplier` in the `geoNear` command with a *spherical* (page 208) example:

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true,
                 distanceMultiplier: 3963.192
               } )
```

The output of the above operation would resemble the following:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 73.46525170413567,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
```

```
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

See also:

The *Distance operator* (page 206).

Querying Haystack Indexes

Haystack indexes are a special `2d` geospatial index that optimized to return results over small areas. To create geospatial indexes see *Haystack Indexes* (page 211).

To query the haystack index, use the `geoSearch` command. You must specify both the coordinate and other field to `geoSearch`, which take the following form:

```
db.runCommand( { geoSearch: <collection>,
                 search: { <field>: <value> } } )
```

For example, to return all documents with the value `restaurants` in the `type` field near the example point, the command would resemble:

```
db.runCommand( { geoSearch: "places",
                 search: { type: "restaurant" },
                 near: [-74, 40.74] } )
```

Note: Haystack indexes are not suited to returning a full list of the closest documents to a particular location, as the closest documents could be far away compared to the `bucketSize`.

Note: *Spherical query operations* (page 208) are not currently supported by haystack indexes.

The `find()` method and `geoNear` command cannot access the haystack index.

6.5 `2d` Geospatial Indexes

6.5.1 Overview

`2d` geospatial indexes make it possible to associate documents with locations in two-dimensional space, such as a point on a map. MongoDB interprets two-dimensional coordinates in a location field as points and can index these points in a special index type to support location-based queries. Geospatial indexes provide special geospatial query operators. For example, you can query for documents based on proximity to another location or based on inclusion in a specified region.

Geospatial indexes support queries on both the coordinate field *and* another field, such as a type of business or attraction. For example, you might write a query to find restaurants a specific distance from a hotel or to find museums within a certain defined neighborhood.

This document describes how to store location data in your documents and how to create geospatial indexes. For information on querying data stored in geospatial indexes, see *Geospatial Queries with 2d Indexes* (page 204).

6.5.2 Store Location Data

To use `2d` geospatial indexes, you must model location data on a predetermined two-dimensional coordinate system, such as longitude and latitude. You store a document's location data as two coordinates in a field that holds either a two-dimensional array or an embedded document with two fields. Consider the following two examples:

```
loc : [ x, y ]
```

```
loc : { x: 1, y: 2 }
```

All documents must store location data in the same order. If you use latitude and longitude as your coordinate system, always store longitude first. MongoDB's *2d spherical index operators* (page 208) only recognize `[ longitude, latitude]` ordering.

6.5.3 Considerations

With the `geoNear` command, a collection can have only one `2d` index. With *geospatial query operators* such as `$near` operator, a collection can have multiple geospatial indexes.

6.5.4 Create a Geospatial Index

To create a geospatial index, use the `ensureIndex` method with the value `2d` for the location field of your collection. Consider the following prototype:

```
db.collection.ensureIndex( { <location field> : "2d" } )
```

MongoDB's *geospatial operations* use this index when querying for location data.

When you create the index, MongoDB converts location data to binary *geohash* values and calculates these values using the location data and the index's location range, as described in *Location Range* (page 210). The default range for `2d` indexes assumes longitude and latitude and uses the bounds -180 inclusive and 180 non-inclusive.

Important: The default boundaries of `2d` indexes 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.

When creating a `2d` index, MongoDB provides the following options:

Location Range

All `2d` geospatial indexes have boundaries defined by a coordinate range. By default, `2d` geospatial indexes assume longitude and latitude have boundaries of -180 inclusive and 180 non-inclusive (i.e. `[-180, 180)`). MongoDB returns an error and rejects documents with coordinate data outside of the specified range.

To build an index with a location range other than the default, use the `min` and `max` options with the `ensureIndex()` operation when creating a `2d` index, as in the following prototype:

```
db.collection.ensureIndex( { <location field>: "2d" } ,
                           { min: <lower bound> , max: <upper bound> } )
```

Location Precision

2d indexes use a *geohash* (page 213) representation of all coordinate data internally. Geohashes have a precision that is determined by the number of bits in the hash. More bits allow the index to provide results with greater precision, while fewer bits mean the index provides results with more limited precision.

Indexes with lower precision have a lower processing overhead for insert operations and will consume less space. However, higher precision indexes means that queries will need to scan smaller portions of the index to return results. The actual stored values are always used in the final query processing, and index precision does not affect query accuracy.

By default, geospatial indexes use 26 bits of precision, which is roughly equivalent to 2 feet or about 60 centimeters of precision using the default range of -180 to 180. You can configure 2d geospatial indexes with up to 32 bits of precision.

To configure a location precision other than the default, use the bits option in the ensureIndex() method, as in the following prototype:

```
db.collection.ensureIndex( {<location field>: "2d"} ,
                           { bits: <bit precision> } )
```

For more information on the relationship between bits and precision, see *Geohash Values* (page 213).

Compound Geospatial Indexes

2d geospatial indexes may be *compound* (page 187), if and only if the field with location data is the first field. A compound geospatial index makes it possible to construct queries that primarily select on a location-based field but also select on a second criteria. For example, you could use such an index to support queries for carpet wholesalers within a specific region.

Note: Geospatial queries will *only* use additional query parameters after applying the geospatial criteria. If your geospatial query criteria selects a large number of documents, the additional query will only filter the result set and *not* result in a more targeted query.

To create a geospatial index with two fields, specify the location field first, then the second field. For example, to create a compound index on the loc location field and on the product field (sorted in ascending order), you would issue the following:

```
db.storeInfo.ensureIndex( { loc: "2d", product: 1 } );
```

This creates an index that supports queries on just the location field (i.e. loc), as well as queries on both the loc and product.

Haystack Indexes

Haystack indexes create "buckets" of documents from the same geographic area in order to improve performance for queries limited to that area.

Each bucket in a haystack index contains all the documents within a specified proximity to a given longitude and latitude. Use the bucketSize parameter of ensureIndex() to determine proximity. A bucketSize of 5 creates an index that groups location values that are within 5 units of the specified longitude and latitude.

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 of a two-dimensional space. Furthermore, the areas defined by buckets can overlap. As a result a document can exist in multiple buckets.

To build a haystack index, use the `bucketSize` parameter in the `ensureIndex()` method, as in the following prototype:

```
db.collection.ensureIndex({ <location field>: "geoHaystack", type: 1 },
                          { bucketSize: <bucket value> })
```

Example

Consider a collection with documents that contain fields similar to the following:

```
{ _id : 100, pos: { long : 126.9, lat : 35.2 }, type : "restaurant"}
{ _id : 200, pos: { long : 127.5, lat : 36.1 }, type : "restaurant"}
{ _id : 300, pos: { long : 128.0, lat : 36.7 }, type : "national park"}
```

The following operations creates a haystack index with buckets that store keys within 1 unit of longitude or latitude.

```
db.collection.ensureIndex( { pos : "geoHaystack", type : 1 }, { bucketSize : 1 } )
```

Therefore, this index stores the document with an `_id` field that has the value `200` in two different buckets:

1. in a bucket that includes the document where the `_id` field has a value of `100`, and
2. 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. For command details, see *Querying Haystack Indexes* (page 209).

Haystack indexes are ideal for returning documents based on location *and* an exact match on a *single* additional criteria. These indexes are not necessarily suited to returning the closest documents to a particular location.

Spherical queries (page 208) are not supported by geospatial haystack indexes.

By default, queries that use a haystack index return 50 documents.

6.5.5 Distance Calculation

MongoDB performs distance calculations before performing `2d` geospatial queries. By default, MongoDB uses flat geometry to calculate distances between points. MongoDB also supports distance calculations using spherical geometry, to provide accurate distances for geospatial information based on a sphere or earth.

Spherical Queries Use Radians for Distance

For spherical operators to function properly, you must convert distances to radians, and convert from radians to the distances units used by your application.

To convert:

- *distance to radians*: divide the distance by the radius of the sphere (e.g. the Earth) in the same units as the distance measurement.
- *radians to distance*: multiply the radian measure by the radius of the sphere (e.g. the Earth) in the units system that you want to convert the distance to.

The radius of the Earth is approximately `3963.192` miles or `6378.137` kilometers.

The following query would return documents from the `places` collection within the circle described by the center `[ -74, 40.74 ]` with a radius of `100` miles:

```
db.places.find( { loc: { $within: { $centerSphere: [ [ -74, 40.74 ] ,
                                                     100 / 3963.192 ] } } } )
```

You may also use the `distanceMultiplier` option to the `geoNear` to convert radians in the `mongod` process, rather than in your application code. Please see the *distance multiplier* (page 208) section.

The following spherical `2d` query, returns all documents in the collection `places` within `100` miles from the point `[ -74, 40.74 ]`.

```
db.runCommand( { geoNear: "places",
                 near: [ -74, 40.74 ],
                 spherical: true
               } )
```

The output of the above command would be:

```
{
   // [ ... ]
   "results" : [
      {
         "dis" : 0.01853688938212826,
         "obj" : {
            "_id" : ObjectId( ... )
            "loc" : [
               -73,
               40
            ]
         }
      }
   ],
   "stats" : {
      // [ ... ]
      "avgDistance" : 0.01853688938212826,
      "maxDistance" : 0.01853714811400047
   },
   "ok" : 1
}
```

Warning: Spherical queries that wrap around the poles or at the transition from `-180` to `180` longitude raise an error.

Note: While the default Earth-like bounds for geospatial indexes are between `-180` inclusive, and `180`, valid values for latitude are between `-90` and `90`.

6.5.6 Geohash Values

To create a geospatial index, MongoDB computes the *geohash* value for coordinate pairs within the specified *range* (page 210) and indexes the geohash for that point .

To calculate a geohash value, continuously divide a 2D map into quadrants. Then assign each quadrant a two-bit value. For example, a two-bit representation of four quadrants would be:

```
01  11

00  10
```

These two-bit values (`00`, `01`, `10`, and `11`) represent each of the quadrants and all points within each quadrant. For a geohash with two bits of resolution, all points in the bottom left quadrant would have a geohash of `00`. The top left quadrant would have the geohash of `01`. The bottom right and top right would have a geohash of `10` and `11`, respectively.

To provide additional precision, continue dividing each quadrant into sub-quadrants. Each sub-quadrant would have the geohash value of the containing quadrant concatenated with the value of the sub-quadrant. The geohash for the upper-right quadrant is `11`, and the geohash for the sub-quadrants would be (clockwise from the top left): `1101`, `1111`, `1110`, and `1100`, respectively.

To calculate a more precise geohash, continue dividing the sub-quadrant and concatenate the two-bit identifier for each division. The more "bits" in the hash identifier for a given point, the smaller possible area that the hash can describe and the higher the resolution of the geospatial index.

6.5.7 Geospatial Indexes and Sharding

You *cannot* use a geospatial 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. Your application may query for geospatial data using `geoNear` and `$within`. However, queries using `$near` are not supported for sharded collections.

6.5.8 Multi-location Documents

New in version 2.0: Support for multiple locations in a document.

While `2d` indexes do not support more than one set of coordinates in a document, you can use a *multi-key indexes* (page 189) to store and 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 prototype data model:

```
{
 "_id": ObjectId(...),
 "locs": [
           [ 55.5, 42.3 ],
           [ -74, 44.74 ],
           { "lat": 55.3, "long": 40.2 }
         ]
}
```

The values of the array may either be arrays holding coordinates, as in `[ 55.5, 42.3 ]`, or embedded documents, as in `{ "lat": 55.3, "long": 40.2 }`.

You could then create a geospatial index on the `locs` field, as in the following:

```
db.places.ensureIndex( { "locs": "2d" } )
```

You may also model the location data as a field inside of a sub-document. In this case, the document would contain a field (e.g. `addresses`) that holds an array of documents where each document has a field (e.g. `loc:`) that holds location coordinates. Consider the following prototype data model:

```
{
 "_id": ObjectId(...),
 "name": "...",
 "addresses": [
                {
                 "context": "home",
                 "loc": [ 55.5, 42.3 ]
                },
                {
                 "context": "home",
                 "loc": [ -74, 44.74 ]
                }
```

```
                ]
}
```

You could then create the geospatial index on the `addresses.loc` field as in the following example:

```
db.records.ensureIndex( { "addresses.loc": "2d" } )
```

For documents with multiple coordinate values, queries may return the same document multiple times if more than one indexed coordinate pair satisfies the query constraints. Use the `uniqueDocs` parameter to `geoNear` or the `$uniqueDocs` operator in conjunction with `$within`.

To include the location field with the distance field in multi-location document queries, specify `includeLocs: true` in the `geoNear` command.

CHAPTER 7

Replication

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

For an overview, see *Replica Set Fundamental Concepts* (page 217). To work with members, see *Replica Set Operation and Management* (page 223). To configure deployment architecture, see *Replica Set Architectures and Deployment Patterns* (page 238). To modify read and write operations, see *Replica Set Considerations and Behaviors for Applications and Development* (page 241). For procedures for performing certain replication tasks, see the *list of replication tutorials* (page 259). For documentation of MongoDB's operational segregation capabilities for replica set deployments see the `/data-center-awareness`

This section contains full documentation, tutorials, and pragmatic guides, as well as links to the reference material that describes all aspects of replica sets.

7.1 Replica Set Use and Operation

Consider these higher level introductions to replica sets:

7.1.1 Replica Set Fundamental Concepts

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

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

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

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

A replica set can have up to 12 members, but only 7 members can have votes. For information regarding non-voting members, see *non-voting members* (page 226)

[1] MongoDB also provides conventional master/slave replication. Master/slave replication operates by way of the same mechanism as replica sets, but lacks the automatic failover capabilities. While replica sets are the recommended solution for production, a replica set can support only 12 members in total. If your deployment requires more than 11 *slave* members, you'll need to use master/slave replication.

See also:

The *Replication* (page 217) index for a list of the documents in this manual that describe the operation and use of replica sets.

Member Configuration Properties

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

- **Secondary-Only**: These members have data but cannot become primary under any circumstance. See *Secondary-Only Members* (page 224).
- **Hidden**: These members are invisible to client applications. See *Hidden Members* (page 224).
- **Delayed**: These members apply operations from the primary's *oplog* after a specified delay. You can think of a delayed member as a form of "rolling backup." See *Delayed Members* (page 225).
- **Arbiters**: These members have no data and exist solely to participate in *elections* (page 218). See *Arbiters* (page 226).
- **Non-Voting**: These members do not vote in elections. Non-voting members are only used for larger sets with more than 7 members. See *Non-Voting Members* (page 226).

For more information about each member configuration, see the *Member Configurations* (page 223) section in the *Replica Set Operation and Management* (page 223) document.

Failover

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

For a detailed explanation of failover, see the *Failover* (page 218) section in the *Replica Set Operation and Management* (page 223) document.

Elections

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

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

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

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

For more information on elections and failover, see:

- The *Failover* (page 218) section in the *Replica Set Operation and Management* (page 223) document.
- The *Election Internals* (page 251) section in the *Replica Set Internals and Behaviors* (page 250) document

Member Priority

In a replica set, every member has a "priority," that helps determine eligibility for *election* (page 218) to *primary*. By default, all members have a priority of `1`, unless you modify the `priority` value. All members have a single vote in elections.

Warning: Always configure the `priority` value to control which members will become primary. Do not configure `votes` except to permit more than 7 secondary members.

For more information on member priorities, see the *Adjusting Priority* (page 228) section in the *Replica Set Operation and Management* (page 223) document.

Consistency

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

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

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

This behavior is sometimes characterized as *eventual consistency* because the secondary member's state will *eventually* reflect the primary's state and MongoDB cannot guarantee *strict consistency* for read operations from secondary members.

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

Rollbacks

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

MongoDB writes the rollback data to a *BSON* file in the database's `dbpath` directory. Use `bsondump` to read the contents of these rollback files and then manually apply the changes to the new primary. There is no way for MongoDB to appropriately and fairly handle rollback situations automatically. Therefore you must intervene manually to apply rollback data. Even after the member completes the rollback and returns to secondary status, administrators will need to apply or decide to ignore the rollback data. MongoDB writes rollback data to a `rollback/` folder within the `dbpath` directory to files with filenames in the following form:

```
<database>.<collection>.<timestamp>.bson
```

For example:

```
records.accounts.2011-05-09T18-10-04.0.bson
```

The best strategy for avoiding all rollbacks is to ensure *write propagation* (page 241) to all or some of the members in the set. Using these kinds of policies prevents situations that might create rollbacks.

Warning: A `mongod` instance will not rollback more than 300 megabytes of data. If your system needs to rollback more than 300 MB, you will need to manually intervene to recover this data. If this is the case, you will find the following line in your `mongod` log:

```
[replica set sync] replSet syncThread: 13410 replSet too much data to roll back
```

In these situations you will need to manually intervene to either save data or to force the member to perform an initial sync from a "current" member of the set by deleting the content of the existing `dbpath` directory.

For more information on failover, see:

- The *Failover and Recovery* (page 236) section in this document.
- The *Failover* (page 218) section in the *Replica Set Operation and Management* (page 223) document.

Application Concerns

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

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

1. *Write Concern* (page 38).

   Write concern sends a MongoDB client a response from the server to confirm successful write operations. In replica sets you can configure *replica acknowledged* (page 38) write concern to ensure that secondary members of the set have replicated operations before the write returns.

2. *Read Preference* (page 243)

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

Read preference and *write concern* have particular *consistency* (page 219) implications.

For a more detailed discussion of application concerns, see *Replica Set Considerations and Behaviors for Applications and Development* (page 241).

Administration and Operations

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

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

- *Replica Set Operation and Management* (page 223)
- *Replica Set Architectures and Deployment Patterns* (page 238)

Oplog

The *oplog* (operations log) is a special *capped collection* that keeps a rolling record of all operations that modify that data stored in your databases. MongoDB applies database operations on the *primary* and then records the operations on the primary's oplog. The *secondary* members then replicate this log and apply the operations to themselves in an asynchronous process. All replica set members contain a copy of the oplog, allowing them to maintain the current state of the database. Operations in the oplog are *idempotent*.

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

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

  If this amount is smaller than a gigabyte, then MongoDB will allocate 1 gigabyte of space.

- For 64-bit OS X systems, MongoDB allocates 183 megabytes of space to the oplog.
- For 32-bit systems, MongoDB allocates about 48 megabytes of space to the oplog.

Before oplog creation, you can specify the size of your oplog with the `oplogSize` option. After you start a replica set member for the first time, you can only change the size of the oplog by using the *Change the Size of the Oplog* (page 271) tutorial.

In most cases, the default oplog size is sufficient. For example, if an oplog that is 5% of free disk space fills up in 24 hours of operations, then secondaries can stop copying entries from the oplog for up to 24 hours without becoming stale. However, most replica sets have much lower operation volumes, and their oplogs can hold a much larger number of operations.

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

- Update operations that affect multiple documents at once.

  The oplog must translate multi-updates into individual operations, in order to maintain *idempotency*. This can use a great deal of oplog space without a corresponding increase in disk utilization.

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

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

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

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

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

To view oplog status, including the size and the time range of operations, issue the `db.printReplicationInfo()` method. For more information on oplog status, see *Check the Size of the Oplog* (page 235).

For additional information about oplog behavior, see *Oplog Internals* (page 250) and *Syncing* (page 252).

Replica Set Deployment

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

The minimum requirements for a replica set include two members with data, for a *primary* and a secondary, and an *arbiter* (page 226). In most circumstances, however, you will want to deploy three data members.

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

Depending on your operational requirements, you may consider adding members configured for a specific purpose including, a *delayed member* to help provide protection against human errors and change control, a *hidden member* to

provide an isolated member for reporting and monitoring, and/or a *secondary only member* (page 224) for dedicated backups.

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

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

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

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

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

Security

In most cases, *replica set* administrators do not have to keep additional considerations in mind beyond the normal security precautions that all MongoDB administrators must take. However, ensure that:

- Your network configuration will allow every member of the replica set to contact every other member of the replica set.
- If you use MongoDB's authentication system to limit access to your infrastructure, ensure that you configure a `keyFile` on all members to permit authentication.

For more information, see the *Security Considerations for Replica Sets* (page 232) section in the *Replica Set Operation and Management* (page 223) document.

Architectures

The architecture and design of the *replica set* deployment can have a great impact on the set's capacity and capability. This section provides a general overview of the architectural possibilities for replica set deployments. However, for most production deployments a conventional 3-member replica set with `priority` values of `1` are sufficient.

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

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

- Ensure that the members of the replica set will always be able to elect a *primary*. Run an odd number of members or run an *arbiter* on one of your application servers if you have an even number of members.
- With geographically distributed members, know where the "quorum" of members will be in the case of any network partitions. Attempt to ensure that the set can elect a primary among the members in the primary data center.
- Consider including a *hidden* (page 224) or *delayed member* (page 225) in your replica set to support dedicated functionality, like backups, reporting, and testing.
- Consider keeping one or two members of the set in an off-site data center, but make sure to configure the *priority* (page 219) to prevent it from becoming primary.
- Create custom write concerns with *replica set tags* to ensure that applications can control the threshold for a successful write operation. Use these write concerns to ensure that operations propagate to specific data centers or to machines of different functions before returning successfully.

For more information regarding replica set configuration and deployments see *Replica Set Architectures and Deployment Patterns* (page 238).

7.1.2 Replica Set Operation and Management

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

See also:

- `rs.status()` (page 286) and `db.isMaster()` (page 286)
- *Replica Set Reconfiguration Process*
- `rs.conf()` (page 286) and `rs.reconfig()` (page 286)
- `http://docs.mongodb.org/manual/reference/replica-configuration`

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

- *Deploy a Replica Set* (page 259)
- *Convert a Standalone to a Replica Set* (page 262)
- *Add Members to a Replica Set* (page 263)
- *Deploy a Geographically Distributed Replica Set* (page 266)
- *Change the Size of the Oplog* (page 271)
- *Force a Member to Become Primary* (page 273)
- *Change Hostnames in a Replica Set* (page 275)
- *Convert a Secondary to an Arbiter* (page 279)
- *Reconfigure a Replica Set with Unavailable Members* (page 281)
- *Recover MongoDB Data following Unexpected Shutdown* (page 283)

Member Configurations

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

Note: A replica set can have up to 12 members, but only 7 members can have votes. For configuration information regarding non-voting members, see *Non-Voting Members* (page 226).

Warning: The `rs.reconfig()` (page 286) shell method can force the current primary to step down, which causes an *election* (page 218). When the primary steps down, the `mongod` closes all client connections. While this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods. To successfully reconfigure a replica set, a majority of the members must be accessible.

See also:

The *Elections* (page 218) section in the *Replica Set Fundamental Concepts* (page 217) document, and the *Election Internals* (page 251) section in the *Replica Set Internals and Behaviors* (page 250) document.

Secondary-Only Members

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

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

To configure a member as secondary-only, set its `priority` value to `0`. Any member with a `priority` equal to `0` will never seek *election* (page 218) and cannot become primary in any situation. For more information on priority levels, see *Member Priority* (page 219).

Note: When updating the replica configuration object, address all members of the set using the index value in the array. 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.

The `_id` rarely corresponds to the array index.

As an example of modifying member priorities, assume a four-member replica set. Use the following sequence of operations in the `mongo` shell to modify member priorities:

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

This reconfigures the set, with the following priority settings:

- Member `0` to a priority of `2` so that it becomes primary, under most circumstances.
- Member `1` to a priority of `1`, which is the default value. Member `1` becomes primary if no member with a *higher* priority is eligible.
- Member `2` to a priority of `0.5`, which makes it less likely to become primary than other members but doesn't prohibit the possibility.
- Member `3` to a priority of `0`. Member `3` cannot become the *primary* member under any circumstances.

Note: If your replica set has an even number of members, add an *arbiter* (page 226) to ensure that members can quickly obtain a majority of votes in an election for primary.

Note: MongoDB does not permit the current *primary* to have a `priority` of `0`. If you want to prevent the current primary from becoming primary, first use `rs.stepDown()` (page 288) to step down the current primary, and then *reconfigure the replica set* with `rs.conf()` (page 286) and `rs.reconfig()` (page 286).

See also:

`priority` and *Replica Set Reconfiguration*.

Hidden Members

Hidden members are part of a replica set but cannot become primary and are invisible to client applications. *However,* hidden members **do** vote in *elections* (page 218).

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

Hidden members have `priority` set `0` and have `hidden` set to `true`.

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

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

After re-configuring the set, the first member of the set in the `members` array will have a priority of `0` so that it cannot become primary. The other members in the set will not advertise the hidden member in the `isMaster` (page 289) or `db.isMaster()` (page 286) output.

Note: You must send the `rs.reconfig()` (page 286) command to a set member that *can* become *primary*. In the above example, if you issue the `rs.reconfig()` (page 286) operation to a member with a `priority` of `0` the operation will fail.

Note: Changed in version 2.0.

For *sharded clusters* running with replica sets before 2.0 if you reconfigured a member as hidden, you *had* to restart `mongos` to prevent queries from reaching the hidden member.

See also:

Replica Set Read Preference (page 243) and *Replica Set Reconfiguration*.

Delayed Members

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

Example

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

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

- Ensure that the length of the delay is equal to or greater than your maintenance windows.
- The size of the oplog is sufficient to capture *more than* the number of operations that typically occur in that period of time. For more information on oplog size, see the *Oplog* (page 220) topic in the *Replica Set Fundamental Concepts* (page 217) document.

Delayed members must have a *priority* set to `0` to prevent them from becoming primary in their replica sets. Also these members should be *hidden* (page 224) to prevent your application from seeing or querying this member.

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

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

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

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

Warning: 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.

See also:

`slaveDelay`, *Replica Set Reconfiguration*, *Oplog* (page 220), *Changing Oplog Size* (page 231) in this document, and the *Change the Size of the Oplog* (page 271) tutorial.

Arbiters

Arbiters are special `mongod` instances that do not hold a copy of the data and thus cannot become primary. Arbiters exist solely to participate in *elections* (page 218).

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

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

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

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

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

- Exchanges of replica set configuration data and of votes. These are not encrypted.

If your MongoDB deployment uses SSL, then all communications between arbiters and the other members of the replica set are secure. See the documentation for *Use MongoDB with SSL Connections* (page 103) for more information. As with all MongoDB components, run arbiters on secure networks.

To add an arbiter, see *Adding an Arbiter* (page 229).

Non-Voting Members

You may choose to change the number of votes that each member has in *elections* (page 218) for *primary*. In general, all members should have only 1 vote to prevent intermittent ties, deadlock, or the wrong members from becoming *primary*. Use *replica set priorities* (page 219) to control which members are more likely to become primary.

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

```
cfg = rs.conf()
cfg.members[3].votes = 0
cfg.members[4].votes = 0
```

```
cfg.members[5].votes = 0
rs.reconfig(cfg)
```

This sequence gives `0` votes to the fourth, fifth, and sixth members of the set according to the order of the `members` array in the output of `rs.conf()` (page 286). This setting allows the set to elect these members as *primary* but does not allow them to vote in elections. If you have three non-voting members, you can add three additional voting members to your set. Place voting members so that your designated primary or primaries can reach a majority of votes in the event of a network partition.

Note: In general and when possible, all members should have only 1 vote. This prevents intermittent ties, deadlocks, or the wrong members from becoming primary. Use *Replica Set Priorities* (page 219) to control which members are more likely to become primary.

See also:

`votes` and *Replica Set Reconfiguration*.

Chained Replication

New in version 2.0.

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.

Beginning with version 2.2.4, you can use the `chainingAllowed` setting in `http://docs.mongodb.org/manual/reference/replica-configuration` to disable chained replication for situations where chained replication is causing lag. For details, see *Chained Replication* (page 227).

Procedures

This section gives overview information on a number of replica set administration procedures. You can find documentation of additional procedures in the *replica set tutorials* (page 259) section.

Adding Members

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

- Make sure the new member's data directory *does not* contain data. The new member will copy the data from an existing member.

  If the new member is in a *recovering* state, it must exit and become a *secondary* before MongoDB can copy all data as part of the replication process. This process takes time but does not require administrator intervention.

- Manually copy the data directory from an existing member. The new member becomes a secondary member and will catch up to the current state of the replica set after a short interval. Copying the data over manually shortens the amount of time for the new member to become current.

  Ensure that you can copy the data directory to the new member and begin replication within the *window allowed by the oplog* (page 220). If the difference in the amount of time between the most recent operation and the most recent operation to the database exceeds the length of the *oplog* on the existing members, then the new instance

will have to perform an initial sync, which completely resynchronizes the data, as described in *Resyncing a Member of a Replica Set* (page 231).

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

For the procedure to add a member to a replica set, see *Add Members to a Replica Set* (page 263).

Removing Members

You may remove a member of a replica set at any time; *however*, for best results always *shut down* the `mongod` instance before removing it from a replica set.

Changed in version 2.2: Before 2.2, you *had* to shut down the `mongod` instance before removing it. While 2.2 removes this requirement, it remains good practice.

To remove a member, use the `rs.remove()` (page 289) method in the `mongo` shell while connected to the current *primary*. Issue the `db.isMaster()` (page 286) command when connected to *any* member of the set to determine the current primary. Use a command in either of the following forms to remove the member:

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

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

You can re-add a removed member to a replica set at any time using the *procedure for adding replica set members* (page 227). Additionally, consider using the *replica set reconfiguration procedure* to change the `host` value to rename a member in a replica set directly.

Replacing a Member

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

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

Use `rs.reconfig()` (page 286) to change the value of the `host` field to reflect the new hostname or port number. `rs.reconfig()` (page 286) will not change the value of `_id`.

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

Warning: Any replica set configuration change can trigger the current *primary* to step down, which forces an *election* (page 218). This causes the current shell session, and clients connected to this replica set, to produce an error even when the operation succeeds.

Adjusting Priority

To change the value of the `priority` in the replica set configuration, use the following sequence of commands in the `mongo` shell:

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

The first operation uses `rs.conf()` (page 286) to set the local variable `cfg` to the contents of the current replica set configuration, which is a *document*. The next three operations change the `priority` value in the `cfg` document for the first three members configured in the `members` array. The final operation calls `rs.reconfig()` (page 286) with the argument of `cfg` to initialize the new configuration.

Note: When updating the replica configuration object, address all members of the set using the index value in the array. 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.

The `_id` rarely corresponds to the array index.

If a member has `priority` set to `0`, it is ineligible to become *primary* and will not seek election. *Hidden members* (page 224), *delayed members* (page 225), and *arbiters* (page 226) all have `priority` set to `0`.

All members have a `priority` equal to `1` by default.

The value of `priority` can be any floating point (i.e. decimal) number between `0` and `1000`. Priorities are only used to determine the preference in election. The priority value is used only in relation to other members. With the exception of members with a priority of `0`, the absolute value of the `priority` value is irrelevant.

Replica sets will preferentially elect and maintain the primary status of the member with the highest `priority` setting.

Warning: Replica set reconfiguration can force the current primary to step down, leading to an election for primary in the replica set. Elections cause the current primary to close all open *client* connections.
Perform routine replica set reconfiguration during scheduled maintenance windows.

See also:

The *Replica Reconfiguration Usage* example revolves around changing the priorities of the `members` of a replica set.

Adding an Arbiter

For a description of *arbiters* and their purpose in *replica sets*, see *Arbiters* (page 226).

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

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

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

   ```
   mkdir /data/arb
   ```

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

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

3. In a `mongo` shell connected to the *primary*, add the arbiter to the replica set by issuing the `rs.addArb()` (page 288) method, which uses the following syntax:

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

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

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

Manually Configure a Secondary's Sync Target

To override the default sync target selection logic, you may manually configure a *secondary* member's sync target for pulling *oplog* entries temporarily. The following operations provide access to this functionality:

- `replSetSyncFrom` (page 293) command, or
- `rs.syncFrom()` (page 289) helper in the `mongo` shell

Only modify the default sync logic as needed, and always exercise caution. `rs.syncFrom()` (page 289) will not affect an in-progress initial sync operation. To affect the sync target for the initial sync, run `rs.syncFrom()` (page 289) operation *before* initial sync.

If you run `rs.syncFrom()` (page 289) during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

Note: `replSetSyncFrom` (page 293) and `rs.syncFrom()` (page 289) provide a temporary override of default behavior. If:

- the `mongod` instance restarts or
- the connection to the sync target closes;

then, the `mongod` instance will revert to the default sync logic and target.

Manage Chained Replication

New in version 2.2.4.

MongoDB enables *chained replication* (page 227) by default. This procedure describes how to disable it and how to re-enable it.

To disable chained replication, set the `chainingAllowed` field in `http://docs.mongodb.org/manual/reference/repl` 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` sub-document. If they do, skip this step.

   > **Warning:** To avoid data loss, skip this step if the configuration settings contain the `settings` sub-document.

   If the current configuration settings **do not** contain the `settings` sub-document, create the sub-document by issuing the following command:

   ```
   cfg.settings = { }
   ```

3. Issue the following sequence of commands to set `chainingAllowed` to `false`:

   ```
   cfg.settings.chainingAllowed = false
   rs.reconfig(cfg)
   ```

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)
```

Note: If chained replication is disabled, you still can use `replSetSyncFrom` (page 293) to specify that a secondary replicates from another secondary. But that configuration will last only until the secondary recalculates which member to sync from.

Changing Oplog Size

The following is an overview of the procedure for changing the size of the oplog. For a detailed procedure, see *Change the Size of the Oplog* (page 271).

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

Resyncing a Member of a Replica Set

When a secondary's replication process falls behind so far that *primary* overwrites oplog entries that the secondary has not yet replicated, that secondary cannot catch up and becomes "stale." When that occurs, you must completely resynchronize the member by removing its data and performing an initial sync.

To do so, use one of the following approaches:

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

  See *Automatically Resync a Stale Member* (page 232).

- Restart the machine with a copy of a recent data directory from another member in the *replica set*. This procedure can replace the data more quickly but requires more manual steps.

  See *Resync by Copying All Datafiles from Another Member* (page 232).

Automatically Resync a Stale Member This procedure relies on MongoDB's regular process for initial sync. This will restore the data on the stale member to reflect the current state of the set. For an overview of MongoDB initial sync process, see the *Syncing* (page 252) section.

To resync the stale member:

1. Stop the stale member's `mongod` instance. On Linux systems you can use *`mongod --shutdown`* Set *`--dbpath`* to the member's data directory, as in the following:

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

2. Delete all data and sub-directories from the member's data directory. By removing the data `dbpath`, MongoDB will perform a complete resync. Consider making a backup first.

3. Restart the `mongod` instance on the member. For example:

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

   At this point, the `mongod` will perform an initial sync. The length of the initial sync may process depends on the size of the database and network connection between members of the replica set.

   Initial sync operations can impact the other members of the set and create additional traffic to the primary, and can only occur if another member of the set is accessible and up to date.

Resync by Copying All Datafiles from Another Member This approach uses a copy of the data files from an existing member of the replica set, or a back of the data files to "seed" the stale member.

The copy or backup of the data files **must** be sufficiently recent to allow the new member to catch up with the *oplog*, otherwise the member would need to perform an initial sync.

Note: In most cases you cannot copy data files from a running `mongod` instance to another, because the data files will change during the file copy operation. Consider the *Backup Strategies for MongoDB Systems* (page 120) documentation for several methods that you can use to capture a consistent snapshot of a running `mongod` instance.

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.

Security Considerations for Replica Sets

In most cases, the most effective ways to control access and to secure the connection between members of a *replica set* depend on network-level access control. Use your environment's firewall and network routing to ensure that traffic *only* from clients and other replica set members can reach your `mongod` instances. If needed, use virtual private networks (VPNs) to ensure secure connections over wide area networks (WANs.)

Additionally, MongoDB provides an authentication mechanism for `mongod` and `mongos` instances connecting to replica sets. These instances enable authentication but specify a shared key file that serves as a shared password.

New in version 1.8: Added support authentication in replica set deployments.

Changed in version 1.9.1: Added support authentication in sharded replica set deployments.

To enable authentication add the following option to your configuration file:

```
keyFile = /srv/mongodb/keyfile
```

Note: You may chose to set these run-time configuration options using the *`--keyFile`* (or *`mongos --keyFile`*) options on the command line.

Setting `keyFile` enables authentication and specifies a key file for the replica set members to use when authenticating to each other. The content of the key file is arbitrary but must be the same on all members of the replica set and on all `mongos` instances that connect to the set.

The key file must be less one kilobyte in size and may only contain characters in the base64 set. The key file must not have group or "world" permissions on UNIX systems. Use the following command to use the OpenSSL package to generate "random" content for use in a key file:

```
openssl rand -base64 753
```

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

Troubleshooting Replica Sets

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

See also:

Monitoring Database Systems (page 109).

Check Replica Set Status

To display the current state of the replica set and current state of each member, run the `rs.status()` (page 286) method in a `mongo` shell connected to the replica set's *primary*. For descriptions of the information displayed by `rs.status()` (page 286), see `http://docs.mongodb.org/manual/reference/replica-status`.

Note: The `rs.status()` (page 286) method is a wrapper that runs the `replSetGetStatus` (page 291) database command.

Check the Replication Lag

Replication lag is a delay between an operation on the *primary* and the application of that operation from the *oplog* to the *secondary*. Replication lag can be a significant issue and can seriously affect MongoDB *replica set* deployments. Excessive replication lag makes "lagged" members ineligible to quickly become primary and increases the possibility that distributed read operations will be inconsistent.

To check the current length of replication lag:

- In a `mongo` shell connected to the primary, call the `db.printSlaveReplicationInfo()` method.

  The returned document displays the `syncedTo` value for each member, which shows you when each member last read from the oplog, as shown in the following example:

  ```
  source:   m1.example.net:30001
      syncedTo: Tue Oct 02 2012 11:33:40 GMT-0400 (EDT)
          = 7475 secs ago (2.08hrs)
  source:   m2.example.net:30002
      syncedTo: Tue Oct 02 2012 11:33:40 GMT-0400 (EDT)
          = 7475 secs ago (2.08hrs)
  ```

  Note: The `rs.status()` (page 286) method is a wrapper around the `replSetGetStatus` (page 291) database command.

- Monitor the rate of replication by watching the oplog time in the "replica" graph in the MongoDB Monitoring Service[2]. For more information see the documentation for MMS[3].

Possible causes of replication lag include:

- **Network Latency**

  Check the network routes between the members of your set to ensure that there is no packet loss or network routing issue.

  Use tools including `ping` to test latency between set members and `traceroute` to expose the routing of packets network endpoints.

- **Disk Throughput**

  If the file system and disk device on the secondary is unable to flush data to disk as quickly as the primary, then the secondary will have difficulty keeping state. Disk-related issues are incredibly prevalent on multi-tenant systems, including vitalized instances, and can be transient if the system accesses disk devices over an IP network (as is the case with Amazon's EBS system.)

  Use system-level tools to assess disk status, including `iostat` or `vmstat`.

- **Concurrency**

  In some cases, long-running operations on the primary can block replication on secondaries. For best results, configure *write concern* (page 38) to require confirmation of replication to secondaries, as described in *Write Concern* (page 241). This prevents write operations from returning if replication cannot keep up with the write load.

  Use the *database profiler* to see if there are slow queries or long-running operations that correspond to the incidences of lag.

- **Appropriate Write Concern**

  If you are performing a large data ingestion or bulk load operation that requires a large number of writes to the primary, particularly with *unacknowledged write concern* (page 38), 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 38) after every 100, 1,000, or an another interval to provide an opportunity for secondaries to catch up with the primary.

  For more information see:

  - *Write Concern* (page 241)
  - *Oplog* (page 220)

Test Connections Between all Members

All members of a *replica set* must be able to connect to every other member of the set to support replication. Always verify connections in both "directions." Networking topologies and firewall configurations prevent normal and required connectivity, which can block replication.

Consider the following example of a bidirectional test of networking:

Example

Given a replica set with three members running on three separate hosts:

- `m1.example.net`

[2] http://mms.mongodb.com/
[3] http://mms.mongodb.com/help/

- `m2.example.net`
- `m3.example.net`

1. Test the connection from `m1.example.net` to the other hosts with the following operation set `m1.example.net`:

   ```
   mongo --host m2.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

2. Test the connection from `m2.example.net` to the other two hosts with the following operation set from `m2.example.net`, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m3.example.net --port 27017
   ```

   You have now tested the connection between `m2.example.net` and `m1.example.net` in both directions.

3. Test the connection from `m3.example.net` to the other two hosts with the following operation set from the `m3.example.net` host, as in:

   ```
   mongo --host m1.example.net --port 27017

   mongo --host m2.example.net --port 27017
   ```

If any connection, in any direction fails, check your networking and firewall configuration and reconfigure your environment to allow these connections.

Check the Size of the Oplog

A larger *oplog* can give a replica set a greater tolerance for lag, and make the set more resilient.

To check the size of the oplog for a given *replica set* member, connect to the member in a `mongo` shell and run the `db.printReplicationInfo()` method.

The output displays the size of the oplog and the date ranges of the operations contained in the oplog. In the following example, the oplog is about 10MB and is able to fit about 26 hours (94400 seconds) of operations:

```
configured oplog size:   10.10546875MB
log length start to end: 94400 (26.22hrs)
oplog first event time:  Mon Mar 19 2012 13:50:38 GMT-0400 (EDT)
oplog last event time:   Wed Oct 03 2012 14:59:10 GMT-0400 (EDT)
now:                     Wed Oct 03 2012 15:00:21 GMT-0400 (EDT)
```

The oplog should be long enough to hold all transactions for the longest downtime you expect on a secondary. At a minimum, an oplog should be able to hold minimum 24 hours of operations; however, many users prefer to have 72 hours or even a week's work of operations.

For more information on how oplog size affects operations, see:

- The *Oplog* (page 220) topic in the *Replica Set Fundamental Concepts* (page 217) document.
- The *Delayed Members* (page 225) topic in this document.
- The *Check the Replication Lag* (page 233) topic in this document.

Note: You normally want the oplog to be the same size on all members. If you resize the oplog, resize it on all members.

To change oplog size, see *Changing Oplog Size* (page 231) in this document or see the *Change the Size of the Oplog* (page 271) tutorial.

Failover and Recovery

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

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

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

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

In many senses, *rollbacks* (page 219) represent a graceful recovery from an impossible failover and recovery situation.

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

You can prevent rollbacks using a *replica acknowledged* (page 38) write concern. These write operations require not only the *primary* to acknowledge the write operation, sometimes even the majority of the set to confirm the write operation before returning.

enabling *write concern*.

See also:

The *Elections* (page 218) section in the *Replica Set Fundamental Concepts* (page 217) document, and the *Election Internals* (page 251) section in the *Replica Set Internals and Behaviors* (page 250) document.

Oplog Entry Timestamp Error

Consider the following error in `mongod` output and logs:

```
replSet error fatal couldn't query the local local.oplog.rs collection.  Terminating mongod after 30
<timestamp> [rsStart] bad replSet oplog entry?
```

Often, an incorrectly typed value in the `ts` field in the last *oplog* entry causes this error. The correct data type is Timestamp.

Check the type of the `ts` value using the following two queries against the oplog collection:

```
db = db.getSiblingDB("local")
db.oplog.rs.find().sort({$natural:-1}).limit(1)
db.oplog.rs.find({ts:{$type:17}}).sort({$natural:-1}).limit(1)
```

The first query returns the last document in the oplog, while the second returns the last document in the oplog where the `ts` value is a Timestamp. The `$type` operator allows you to select *BSON type* 17, is the Timestamp data type.

If the queries don't return the same document, then the last document in the oplog has the wrong data type in the `ts` field.

Example

If the first query returns this as the last oplog entry:

```
{ "ts" : {t: 1347982456000, i: 1},
  "h" : NumberLong("8191276672478122996"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 4 } }
```

And the second query returns this as the last entry where `ts` has the `Timestamp` type:

```
{ "ts" : Timestamp(1347982454000, 1),
  "h" : NumberLong("6188469075153256465"),
  "op" : "n",
  "ns" : "",
  "o" : { "msg" : "Reconfig set", "version" : 3 } }
```

Then the value for the `ts` field in the last oplog entry is of the wrong data type.

To set the proper type for this value and resolve this issue, use an update operation that resembles the following:

```
db.oplog.rs.update( { ts: { t:1347982456000, i:1 } },
                    { $set: { ts: new Timestamp(1347982456000, 1)}})
```

Modify the timestamp values as needed based on your oplog entry. This operation may take some period to complete because the update must scan and pull the entire oplog into memory.

Duplicate Key Error on `local.slaves`

The *duplicate key on local.slaves* error, occurs when a *secondary* or *slave* changes its hostname and the *primary* or *master* tries to update its `local.slaves` collection with the new name. The update fails because it contains the same `_id` value as the document containing the previous hostname. The error itself will resemble the following.

```
exception 11000 E11000 duplicate key error index: local.slaves.$_id_  dup key: { : ObjectId('<object
```

This is a benign error and does not affect replication operations on the *secondary* or *slave*.

To prevent the error from appearing, drop the `local.slaves` collection from the *primary* or *master*, with the following sequence of operations in the `mongo` shell:

```
use local
db.slaves.drop()
```

The next time a *secondary* or *slave* polls the *primary* or *master*, the *primary* or *master* recreates the `local.slaves` collection.

Elections and Network Partitions

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

That means that if a primary steps down and neither side of the partition has a majority on its own, the set will not elect a new primary and the set will become read only. To avoid this situation, attempt to place a majority of instances in one data center with a minority of instances in a secondary facility.

See

Election Internals (page 251).

7.1.3 Replica Set Architectures and Deployment Patterns

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

For an overview of operational practices and background information, see the *Architectures* (page 222) topic in the *Replica Set Fundamental Concepts* (page 217) document.

Three Member Sets

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

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

  This makes *failover* (page 236) possible and ensures there exists two full and independent copies of the data set at all times. If the primary fails, the replica set elects another member as primary and continues replication until the primary recovers.

Note: While not *recommended*, the minimum *supported* configuration for replica sets includes one *primary*, one *secondary*, and one *arbiter* (page 226). The arbiter requires fewer resources and lowers costs but sacrifices operational flexibility and redundancy.

See also:

Deploy a Replica Set (page 259).

Sets with Four or More Members

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

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

- The set has an odd number of voting members.

  If you have an *even* number of voting members, deploy an *arbiter* (page 226) to create an odd number.

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

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

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

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

See also:

Add Members to a Replica Set (page 263).

Geographically Distributed Sets

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

In many circumstances, these deployments consist of the following:

- One *primary* in the first (i.e., primary) data center.
- One *secondary* member in the primary data center. This member can become the primary member at any time.
- One secondary member in a secondary data center. This member is ineligible to become primary. Set its `local.system.replset.members[n].priority` to `0`.

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

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

If the primary data center fails, you can manually recover the data set from the secondary data center. With appropriate *write concern* (page 38) there will be no data loss and downtime can be minimal.

When you add a secondary data center, make sure to keep an odd number of members overall to prevent ties during elections for primary by deploying an *arbiter* (page 226) in your primary data center. For example, if you have three members in the primary data center and add a member in a secondary center, you create an even number. To create an odd number and prevent ties, deploy an *arbiter* (page 226) in your primary data center.

See also:

Deploy a Geographically Distributed Replica Set (page 266)

Non-Production Members

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

- **Low-Priority**: These members have `local.system.replset.members[n].priority` settings such that they are either unable to become *primary* or *very* unlikely to become primary. In all other respects these low-priority members are identical to other replica set member. (See: *Secondary-Only Members* (page 224).)
- **Hidden**: These members cannot become primary *and* the set excludes them from the output of `db.isMaster()` (page 286) and from the output of the database command `isMaster` (page 289). Excluding hidden members from such outputs prevents clients and drivers from using hidden members for secondary reads. (See: *Hidden Members* (page 224).)
- **Voting**: This changes the number of votes that a member of the replica set has in elections. In general, use priority to control the outcome of elections, as weighting votes introduces operational complexities and risks. Only modify the number of votes when you need to have more than 7 members in a replica set. (See: *Non-Voting Members* (page 226).)

Note: All members of a replica set vote in elections *except* for *non-voting* (page 226) members. Priority, hidden, or delayed status does not affect a member's ability to vote in an election.

Backups

For some deployments, keeping a replica set member for dedicated backup purposes is operationally advantageous. Ensure this member is close, from a networking perspective, to the primary or likely primary. Ensure that the *replication lag* is minimal or non-existent. To create a dedicated *hidden member* (page 224) for the purpose of creating backups.

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

See also:

Backup Strategies for MongoDB Systems (page 120).

Delayed Replication

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

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

See *Replica Set Delayed Nodes* (page 225) for more information about configuring delayed replica set members.

Reporting

Typically *hidden members* provide a substrate for reporting purposes, because the replica set segregates these instances from the cluster. Since no secondary reads reach hidden members, they receive no traffic beyond what replication requires. While hidden members are not electable as primary, they are still able to *vote* in elections for primary. If your operational parameters requires this kind of reporting functionality, see *Hidden Replica Set Nodes* (page 224) and `local.system.replset.members[n].hidden` for more information regarding this functionality.

Cold Standbys

For some sets, it may not be possible to initialize a new member in a reasonable amount of time. In these situations, it may be useful to maintain a secondary member with an up-to-date copy for the purpose of replacing another member in the replica set. In most cases, these members can be ordinary members of the replica set, but in large sets, with varied hardware availability, or given some patterns of *geographical distribution* (page 239), you may want to use a member with a different *priority*, *hidden*, or voting status.

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

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

See also:

Secondary Only (page 224), and *Hidden Nodes* (page 224).

Arbiters

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

To deploy an arbiter, see the *Arbiters* (page 226) topic in the *Replica Set Operation and Management* (page 223) document.

7.1.4 Replica Set Considerations and Behaviors for Applications and Development

From the perspective of a client application, whether a MongoDB instance is running as a single server (i.e. "standalone") or a *replica set* is transparent. However, replica sets offer some configuration options for write and read operations. [4] This document describes those options and their implications.

Write Concern

MongoDB's built-in *write concern* confirms the success of write operations to a *replica set's primary*. Write concern uses the `getLastError` command after write operations to return an object with error information or confirmation that there are no errors.

After the *driver write concern change* (page 419) all officially supported MongoDB drivers enable write concern by default.

Verify Write Operations

The default write concern confirms write operations only on the primary. You can configure write concern to confirm write operations to additional replica set members as well by issuing the `getLastError` command with the `w` option.

The `w` option confirms that write operations have replicated to the specified number of replica set members, including the primary. You can either specify a number or specify `majority`, which ensures the write propagates to a majority of set members. The following example ensures the operation has replicated to two members (the primary and one other member):

```
db.runCommand( { getLastError: 1, w: 2 } )
```

The following example ensures the write operation has replicated to a majority of the configured members of the set.

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

If you specify a `w` value greater than the number of members that hold a copy of the data (i.e., greater than the number of non-*arbiter* members), the operation blocks until those members become available. This can cause the operation to block forever. To specify a timeout threshold for the `getLastError` operation, use the `wtimeout` argument. The following example sets the timeout to 5000 milliseconds:

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

```
db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )
```

Modify Default Write Concern

You can configure your own "default" `getLastError` behavior for a replica set. Use the `getLastErrorDefaults` setting in the `replica set configuration`. The following sequence of commands creates a configuration that waits for the write operation to complete on a majority of the set members before returning:

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

The `getLastErrorDefaults` setting affects only those `getLastError` commands that have *no* other arguments.

Note: Use of insufficient write concern can lead to *rollbacks* (page 219) in the case of *replica set failover* (page 236). Always ensure that your operations have specified the required write concern for your application.

See also:

Write Concern (page 38) and *connections-write-concern*

Custom Write Concerns

You can use replica set tags to create custom write concerns using the `getLastErrorDefaults` and `getLastErrorModes` replica set settings.

Note: Custom write concern modes specify the field name and a number of *distinct* values for that field. By contrast, read preferences use the value of fields in the tag document to direct read operations.

In some cases, you may be able to use the same tags for read preferences and write concerns; however, you may need to create additional tags for write concerns depending on the requirements of your application.

Single Tag Write Concerns Consider a five member replica set, where each member has one of the following tag sets:

```
{ "use": "reporting" }
{ "use": "backup" }
{ "use": "application" }
{ "use": "application" }
{ "use": "application" }
```

You could create a custom write concern mode that will ensure that applicable write operations will not return until members with two different values of the `use` tag have acknowledged the write operation. Create the mode with the following sequence of operations in the `mongo` shell:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { use2: { "use": 2 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `multiUse` to the `w` option of `getLastError` as follows:

```
db.runCommand( { getLastError: 1, w: use2 } )
```

Specific Custom Write Concerns If you have a three member replica with the following tag sets:

```
{ "disk": "ssd" }
{ "disk": "san" }
{ "disk": "spinning" }
```

You cannot specify a custom `getLastErrorModes` value to ensure that the write propagates to the `san` before returning. However, you may implement this write concern policy by creating the following additional tags, so that the set resembles the following:

```
{ "disk": "ssd" }
{ "disk": "san", "disk.san": "san" }
{ "disk": "spinning" }
```

Then, create a custom `getLastErrorModes` value, as follows:

```
cfg = rs.conf()
cfg.settings = { getLastErrorModes: { san: { "disk.san": 1 } } }
rs.reconfig(cfg)
```

To use this mode pass the string `san` to the `w` option of `getLastError` as follows:

```
db.runCommand( { getLastError: 1, w: san } )
```

This operation will not return until a replica set member with the tag `disk.san` returns.

You may set a custom write concern mode as the default write concern mode using `getLastErrorDefaults` replica set as in the following setting:

```
cfg = rs.conf()
cfg.settings.getLastErrorDefaults = { ssd:1 }
rs.reconfig(cfg)
```

See also:

replica-set-configuration-tag-sets for further information about replica set reconfiguration and tag sets.

Read Preference

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

Background

By default, an application directs its read operations to the *primary* member in a *replica set*. Reading from the primary guarantees that read operations reflect the latest version of a document. However, for an application that does not require fully up-to-date data, you can improve read throughput, or reduce latency, by distributing some or all reads to secondary members of the replica set.

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

- Running systems operations that do not affect the front-end application, operations such as backups and reports.
- Providing low-latency queries for geographically distributed deployments. If one secondary is closer to an application server than the primary, you may see better performance for that application if you use secondary reads.

- Providing graceful degradation in *failover* (page 236) situations where a set has *no* primary for 10 seconds or more. In this use case, you should give the application the `primaryPreferred` (page 244) read preference, which prevents the application from performing reads if the set has no primary.

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

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

If read operations account for a large percentage of your application's traffic, distributing reads to secondary members can improve read throughput. However, in most cases *sharding* (page 295) provides better support for larger scale operations, as clusters can distribute read and write operations across a group of machines.

Read Preference Modes

New in version 2.2.

MongoDB `drivers` support five read preference modes:

- `primary` (page 244)
- `primaryPreferred` (page 244)
- `secondary` (page 245)
- `secondaryPreferred` (page 245)
- `nearest` (page 245)

You can specify a read preference mode on connection objects, database object, collection object, or per-operation. The syntax for specifying the read preference mode is specific to the driver and to the idioms of the host language[5].

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.

Warning: All read preference modes except `primary` (page 244) may return stale data as *secondaries* replicate operations from the primary with some delay.
Ensure that your application can tolerate stale data if you choose to use a non-`primary` (page 244) mode.

For more information, see *read preference background* (page 243) and *read preference behavior* (page 247). See also the documentation for your driver[6].

`primary`
: All read operations use only the current replica set *primary*. This is the default. If the primary is unavailable, read operations produce an error or throw an exception.

  The `primary` (page 244) read preference mode is not compatible with read preference modes that use *tag sets* (page 246). If you specify a tag set with `primary` (page 244), the driver will produce an error.

[5] http://api.mongodb.org/
[6] http://api.mongodb.org/

primaryPreferred

In most situations, operations read from the *primary* member of the set. However, if the primary is unavailable, as is the case during *failover* situations, operations read from secondary members.

When the read preference includes a *tag set* (page 246), 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 244) 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 244) 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 246), 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 248). If no secondaries have matching tags, the read operation produces an error. [7]

Read operations using the `secondary` (page 245) 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 246), 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 248). If no secondaries have matching tags, the read operation produces an error.

Read operations using the `secondaryPreferred` (page 245) mode may return stale data.

nearest

The driver reads from the *nearest* member of the *set* according to the *member selection* (page 248) process. Reads in the `nearest` (page 245) mode do not consider the member's *type*. Reads in `nearest` (page 245) 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 246), 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 248).

Read operations using the `nearest` (page 245) 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 245) mode prefers low latency reads over a member's *primary* or *secondary* status.

[7] If your set has more than one secondary, and you use the `secondary` (page 245) read preference mode, consider the following effect. If you have a *three member replica set* (page 238) with a primary and two secondaries, and if one secondary becomes unavailable, all `secondary` (page 245) 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 245), 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 248) for more information.

Tag Sets

Tag sets allow you to specify custom *read preferences* (page 243) and *write concerns* (page 38) so that your application can target operations to specific members, based on custom parameters.

Note: Consider the following properties of read preferences:

- Custom read preferences and write concerns evaluate tags sets in different ways.
- Read preferences consider the value of a tag when selecting a member to read from.
- Write concerns ignore the value of a tag to when selecting a member *except* to consider whether or not the value is unique.

A tag set for a read operation may resemble the following document:

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

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

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

However, the following tag sets would *not* be able to fulfill this query:

```
{ "disk": "ssd" }
{ "use": "reporting" }
{ "disk": "ssd", "use": "production" }
{ "disk": "ssd", "use": "production", "rack": 3 }
{ "disk": "spinning", "use": "reporting", "mem": "32" }
```

Therefore, tag sets make it possible to ensure that read operations target specific members in a particular data center or `mongod` instances designated for a particular class of operations, such as reporting or analytics. For information on configuring tag sets, see *replica-set-configuration-tag-sets* in the `http://docs.mongodb.org/manual/reference/replica-configuration` document. You can specify tag sets with the following read preference modes:

- `primaryPreferred` (page 244)
- `secondary` (page 245)
- `secondaryPreferred` (page 245)
- `nearest` (page 245)

You cannot specify tag sets with the `primary` (page 244) read preference mode.

Tags are not compatible with `primary` (page 244) and only apply when *selecting* (page 248) a *secondary* member of a set for a read operation. However, the `nearest` (page 245) read mode, when combined with a tag set will select the nearest member that matches the specified tag set, which may be a primary or secondary.

All interfaces use the same *member selection logic* (page 248) to choose the member to which to direct read operations, basing the choice on read preference mode and tag sets.

For more information on how read preference *modes* (page 244) interact with tag sets, see the documentation for each read preference mode.

Behavior

Changed in version 2.2.

Auto-Retry Connection between MongoDB drivers and `mongod` instances in a *replica set* must balance two concerns:

1. The client should attempt to prefer current results, and any connection should read from the same member of the replica set as much as possible.
2. The client should minimize the amount of time that the database is inaccessible as the result of a connection issue, networking problem, or *failover* in a replica set.

As a result, MongoDB drivers and `mongos`:

- Reuse a connection to specific `mongod` for as long as possible after establishing a connection to that instance. This connection is *pinned* to this `mongod`.
- Attempt to reconnect to a new member, obeying existing *read preference modes* (page 244), 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 244) and *tag set* (page 246). 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 244).
- After detecting a failover situation, [8] the driver attempts to refresh the state of the replica set as quickly as possible.

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

- The application performs a read with a different read preference.
- The thread terminates.
- The client receives a socket exception, as is the case when there's a network error or when the `mongod` closes connections during a *failover*. This triggers a *retry* (page 247), which may be transparent to the application.

If an application thread issues a query with the `primaryPreferred` (page 244) mode while the primary is inaccessible, the thread will carry the association with that secondary for the lifetime of the thread. The thread will associate with the primary, if available, only after issuing a query with a different read preference, even if a primary becomes available. By extension, if a thread issues a read with the `secondaryPreferred` (page 245) when all secondaries

[8] When a *failover* occurs, all members of the set close all client connections that produce a socket error in the driver. This behavior prevents or minimizes *rollback*.

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

Member Selection Clients, by way of their drivers, and `mongos` instances for sharded clusters periodically update their view of the replica set's state: which members are up or down, which member is primary, and the latency to each `mongod` instance.

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

1. Assembles a list of suitable members, taking into account member type (i.e. secondary, primary, or all members.)
2. Excludes members not matching the tag sets, if specified.
3. Determines which suitable member is the closest to the client in absolute terms.
4. Builds a list of members that are within a defined ping distance (in milliseconds) of the "absolute nearest" member. [9]
5. Selects a member from these hosts at random. The member receives the read operation.

Once the application selects a member of the set to use for read operations, the driver continues to use this connection for read preference until the application specifies a new read preference or something interrupts the connection. See *Request Association* (page 247) for more information.

Sharding and `mongos` Changed in version 2.2: Before version 2.2, `mongos` did not support the *read preference mode semantics* (page 244).

In most *sharded clusters*, a *replica set* provides each shard where read preferences are also applicable. Read operations in a sharded cluster, with regard to read preference, are identical to unsharded replica sets.

Unlike simple replica sets, in sharded clusters, all interactions with the shards pass from the clients to the `mongos` instances that are actually connected to the set members. `mongos` is responsible for the application of the read preferences, which is transparent to applications.

There are no configuration changes required for full support of read preference modes in sharded environments, as long as the `mongos` 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 244), the default, unless, the `mongos` reuses an existing connection that has a different mode set.

  Always explicitly set your read preference mode to prevent confusion.

- All `nearest` (page 245) 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.

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 244) for the following commands:

- `group`
- `mapReduce` [10]

[9] Applications can configure the threshold used in this stage. The default "acceptable latency" is 15 milliseconds, which you can override in the drivers with their own `secondaryAcceptableLatencyMS` option. For `mongos` you can use the `--localThreshold` or `localThreshold` runtime options to set this value.

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

- aggregate
- collStats
- dbStats
- count
- distinct
- geoNear
- geoSearch
- geoWalk

mongos currently does not route commands using read preferences; clients send all commands to shards' primaries. See SERVER-7423[11].

Uses for non-Primary Read Preferences

You must exercise care when specifying read preferences: modes other than primary (page 244) can *and will* return stale data. These secondary queries will not include the most recent write operations to the replica set's *primary*. Nevertheless, there are several common use cases for using non-primary (page 244) read preference modes:

- Reporting and analytics workloads.

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

  Also consider using secondary (page 245) in conjunction with a direct connection to a *hidden member* (page 224) of the set.

- Providing local reads for geographically distributed applications.

  If you have application servers in multiple data centers, you may consider having a *geographically distributed replica set* (page 239) and using a non primary read preference or the nearest (page 245) to avoid network latency.

- Maintaining availability during a failover.

  Use primaryPreferred (page 244) if you want your application to do consistent reads from the primary under normal circumstances, but to allow stale reads from secondaries in an emergency. This provides a "read-only mode" for your application during a failover.

Warning: In some situations using secondaryPreferred (page 245) to distribute read load to replica sets may carry significant operational risk: if all secondaries are unavailable and your set has enough *arbiters* to prevent the primary from stepping down, then the primary will receive all traffic from clients.
For this reason, use secondary (page 245) to distribute read load to replica sets, not secondaryPreferred (page 245).

Using read modes other than primary (page 244) and primaryPreferred (page 244) to provide extra capacity is not in and of itself justification for non-primary (page 244) in many cases. Furthermore, *sharding* (page 295) increases read and write capacity by distributing read and write operations across a group of machines.

[11] https://jira.mongodb.org/browse/SERVER-7423

7.1.5 Replica Set Internals and Behaviors

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

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

- `http://docs.mongodb.org/manual/reference/local-database`
- *Replica Set Commands* (page 286)
- `http://docs.mongodb.org/manual/reference/replication-info`
- `http://docs.mongodb.org/manual/reference/replica-configuration`

Oplog Internals

For an explanation of the oplog, see *Oplog* (page 220).

Under various exceptional situations, updates to a *secondary's* oplog might lag behind the desired performance time. See *Replication Lag* (page 233) for details.

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

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

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

Read Preference Internals

MongoDB uses *single-master replication* to ensure that the database remains consistent. However, clients may modify the *read preferences* (page 243) on a per-connection basis in order to distribute read operations to the *secondary* members of a *replica set*. Read-heavy deployments may achieve greater query throughput by distributing reads to secondary members. But keep in mind that replication is asynchronous; therefore, reads from secondaries may not always reflect the latest writes to the *primary*.

See also:

Consistency (page 219)

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

Member Configurations

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

- *Secondary-only* (page 224) members have their `priority` values set to `0` and thus are not eligible for election as primaries.
- *Hidden* (page 224) members do not appear in the output of `db.isMaster()` (page 286). This prevents clients from discovering and potentially querying the member in question.

- *Delayed* (page 225) members lag a fixed period of time behind the primary. These members are typically used for disaster recovery scenarios. For example, if an administrator mistakenly truncates a collection, and you discover the mistake within the lag window, then you can manually fail over to the delayed member.
- *Arbiters* (page 226) exist solely to participate in elections. They do not replicate data from the primary.

In almost every case, replica sets simplify the process of administering database replication. However, replica sets still have a unique set of administrative requirements and concerns. Choosing the right *system architecture* (page 238) for your data set is crucial.

See also:

The *Member Configurations* (page 223) topic in the *Replica Set Operation and Management* (page 223) document.

Security Internals

Administrators of replica sets also have unique *monitoring* (page 115) and *security* (page 232) concerns. The *replica set functions* in the `mongo` shell, provide the tools necessary for replica set administration. In particular use the `rs.conf()` (page 286) to return a *document* that holds the `replica set configuration` and use `rs.reconfig()` (page 286) to modify the configuration of an existing replica set.

Election Internals

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

The following events can trigger an election:

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

In an election, all members have one vote, including *hidden* (page 224) members, *arbiters* (page 226), and even recovering members. Any `mongod` can veto an election.

In the default configuration, all members have an equal chance of becoming primary; however, it's possible to set `priority>` values that weight the election. In some architectures, there may be operational reasons for increasing the likelihood of a specific replica set member becoming primary. For instance, a member located in a remote data center should *not* become primary. See: *Member Priority* (page 219) for more information.

Any member of a replica set can veto an election, even if the member is a *non-voting member* (page 226).

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

- If the member seeking an election is not a member of the voter's set.
- If the member seeking an election is not up-to-date with the most recent operation accessible in the replica set.
- If the member seeking an election has a lower priority than another member in the set that is also eligible for election.

- If a *secondary only member* (page 224) [12] is the most current member at the time of the election, another eligible member of the set will catch up to the state of this secondary member and then attempt to become primary.
- If the current primary member has more recent operations (i.e. a higher "optime") than the member seeking election, from the perspective of the voting member.
- The current primary will veto an election if it has the same or more recent operations (i.e. a "higher or equal optime") than the member seeking election.

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

- Replica set members send heartbeats (pings) to each other every 2 seconds. If a heartbeat does not return for more than 10 seconds, the other members mark the delinquent member as inaccessible.
- Replica set members compare priorities only with other members of the set. The absolute value of priorities does not have any impact on the outcome of replica set elections, with the exception of the value `0`, which indicates the member cannot become primary and cannot seek election. For details, see *Adjusting Priority* (page 228).
- A replica set member cannot become primary *unless* it has the highest "optime" of any visible member in the set.
- If the member of the set with the highest priority is within 10 seconds of the latest *oplog* entry, then the set will *not* elect a primary until the member with the highest priority catches up to the latest operation.

See also:

Non-voting members in a replica set (page 226), *Adjusting Priority* (page 228), and `replica configuration`.

Syncing

In order to remain up-to-date with the current state of the *replica set*, set members *sync*, or copy, *oplog* entries from other members. Members sync data at two different points:

- *Initial sync* occurs when MongoDB creates new databases on a new or restored member, populating the member with the replica set's data. When a new or restored member joins or rejoins a set, the member waits to receive heartbeats from other members. By default, the member syncs from the *closest* member of the set that is either the primary or another secondary with more recent oplog entries. This prevents two secondaries from syncing from each other.
- *Replication* occurs continually after initial sync and keeps the member updated with changes to the replica set's data.

In MongoDB 2.0, secondaries only change sync targets if the connection to the sync target drops [13] or produces an error.

For example:

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

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

[12] Remember that *hidden* (page 224) and *delayed* (page 225) imply *secondary-only* (page 224) configuration.

[13] Secondaries will stop syncing from a member if the connection used to poll oplog entries is unresponsive for 30 seconds. If a connection times out, the member may select a new member to sync from. Before version 2.2, secondaries would way 10 minutes to select a new member to sync from.

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

In MongoDB 2.2, secondaries also use the following additional sync behaviors:

- Secondaries will sync from *delayed members* (page 225) *only* if no other member is available.
- Secondaries will *not* sync from *hidden members* (page 224).
- Secondaries will *not* start syncing from a member in a *recovering* state.
- For one member to sync from another, both members must have the same value, either `true` or `false`, for the `buildIndexes` field.

Multithreaded Replication

MongoDB applies write operations in batches using a multithreaded approach. The replication process divides each batch among a group of threads which apply many operations with greater concurrency.

Even though threads may apply operations out of order, a client reading data from a secondary will never return documents that reflect an in-between state that never existed on the primary. To ensure this consistency, MongoDB blocks all read operations while applying the batch of operations.

To help improve the performance of operation application, MongoDB fetches all the memory pages that hold data and indexes that the operations in the batch will affect. The prefetch stage minimizes the amount of time MongoDB must hold the write lock to apply operations. See the `replIndexPrefetch` setting to modify the index fetching behavior.

Pre-Fetching Indexes to Improve Replication Throughput

By default, secondaries will in most cases pre-fetch *Indexes* (page 185) associated with the affected document to improve replication throughput.

You can limit this feature to pre-fetch only the index on the `_id` field, or you can disable this feature entirely. For more information, see `replIndexPrefetch`.

The following document describes master-slave replication, which is deprecated. Use replica sets instead of master-slave in all new deployments.

7.1.6 Master Slave Replication

Important: *Replica sets* (page 217) 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*.

Replica sets provide functional super-set of master-slave and are more robust for production use. Master-slave replication preceded replica and makes it possible have a large number of non-master (i.e. slave) and to *only* replicate operations for a single database; however, master-slave replication provides less redundancy, and does not automate failover. See *Deploy Master-Slave Equivalent using Replica Sets* (page 256) for a replica set configuration that is equivalent to master-slave replication.

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` 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` collection.

Configuration Options for Master-Slave Deployments

As an alternative to specifying the *`--source`* run-time option, can add a document to `local.sources` 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` collection. The model of the `local.sources` document is as follows:

`host`
: The host field specifies the `mastermongod` instance, and holds a resolvable hostname, i.e. IP address, or a name from a `host` file, or preferably a fully qualified domain name.

  You can append `<:port>` to the host name if the `mongod` is not running on the default `27017` port.

`only`
: Optional. Specify a name of a database. When specified, MongoDB will only replicate the indicated database.

Operational Considerations for Replication with Master Slave Deployments

Master instances store operations in an *oplog* which is a `capped collection`. As a result, if a slave falls too far behind the state of the master, it cannot "catchup" and must re-sync from scratch. Slave may become out of sync with a master if:

- The slave falls far behind the data updates available from that master.
- The slave stops (i.e. shuts down) and restarts later after the master has overwritten the relevant operations from the master.

When slaves, are out of sync, replication stops. Administrators must intervene manually to restart replication. Use the `resync` (page 290) command. Alternatively, the *--autoresync* allows a slave to restart replication automatically, after ten second pause, when the slave falls out of sync with the master. With *--autoresync* specified, the slave will only attempt to re-sync once in a ten minute period.

To prevent these situations you should specify a larger oplog when you start the `master` instance, by adding the *--oplogSize* option when starting `mongod`. If you do not specify *--oplogSize*, `mongod` will allocate 5% of available disk space on start up to the oplog, with a minimum of 1GB for 64bit machines and 50MB for 32bit machines.

Run time Master-Slave Configuration

MongoDB provides a number of run time configuration options for `mongod` instances in *master-slave* deployments. You can specify these options in *configuration files* (page 95) or on the command-line. See documentation of the following:

- For *master* nodes:
  - `master`
  - `slave`
- For *slave* nodes:
  - `source`
  - `only`
  - `slaveDelay`

Also consider the *Master-Slave Replication Command Line Options* for related options.

Diagnostics

On a *master* instance, issue the following operation in the `mongo` shell to return replication status from the perspective of the master:

```
db.printReplicationInfo()
```

On a *slave* instance, use the following operation in the `mongo` shell to return the replication status from the perspective of the slave:

```
db.printSlaveReplicationInfo()
```

Use the `serverStatus` as in the following operation, to return status of the replication:

```
db.serverStatus()
```

See *server status repl fields* for documentation of the relevant section of output.

Security

When running with `auth` enabled, in *master-slave* deployments, you must create a user account for the local database on both `mongod` instances. Log in, and authenticate to the `admin` database on the *slave* instance, and then create the `repl` user on the `local` database, with the following operation:

```
use local
db.addUser('repl', <replpassword>)
```

Once created, repeat the operation on the *master* instance.

The slave instance first looks for a user named `repl` in the `local.system.users` collection. If present, the slave uses this user account to authenticate to the `local` database in the `master` instance. If the `repl` user does not exist, the slave instance attempts to authenticate using the first user document in the `local.system.users` collection.

The `local` database works like the `admin` database: an account for local has access to the entire server.

See also:

Security (page 133) 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>` [14] provide replication that is roughly equivalent to a two-instance master-slave deployment:

```
{
  _id : 'setName',
  members : [
    { _id : 0, host : "<master>", priority : 1 },
    { _id : 1, host : "<slave>", priority : 0, votes : 0 }
  ]
}
```

See `http://docs.mongodb.org/manual/reference/replica-configuration` for more information about replica set configurations.

Failing over to a Slave (Promotion)

To permanently failover from a unavailable or damaged *master* (`A` in the following example) to a *slave* (`B`):

1. Shut down `A`.
2. Stop `mongod` on `B`.
3. Back up and move all data files that begin with `local` on `B` from the `dbpath`.

   Warning: Removing `local.*` is irrevocable and cannot be undone. Perform this step with extreme caution.

4. Restart `mongod` on `B` with the *`--master`* option.

Note: This is a one time operation, and is not reversible. `A` cannot become a slave of `B` until it completes a full resync.

Inverting Master and Slave

If you have a *master* (`A`) and a *slave* (`B`) and you would like to reverse their roles, follow this procedure. The procedure assumes `A` is healthy, up-to-date and available.

[14] In replica set configurations, the `host` field must hold a resolvable hostname.

If A is not healthy but the hardware is okay (power outage, server crash, etc.), skip steps 1 and 2 and in step 8 replace all of A's files with B's files in step 8.

If A is not healthy and the hardware is not okay, replace A with a new machine. Also follow the instructions in the previous paragraph.

To invert the master and slave in a deployment:

1. Halt writes on A using the *fsync* command.
2. Make sure B is up to date with the state of A.
3. Shut down B.
4. Back up and move all data files that begin with local on B from the dbpath to remove the existing local.sources data.

   Warning: Removing local.* is irrevocable and cannot be undone. Perform this step with extreme caution.

5. Start B with the *--master* option.
6. Do a write on B, which primes the *oplog* to provide a new sync start point.
7. Shut down B. B will now have a new set of data files that start with local.
8. Shut down A and replace all files in the dbpath of A that start with local with a copy of the files in the dbpath of B that begin with local.

   Considering compressing the local files from B while you copy them, as they may be quite large.

9. Start B with the *--master* option.
10. Start A with all the usual slave options, but include *fastsync*.

Creating a Slave from an Existing Master's Disk Image

If you can stop write operations to the *master* for an indefinite period, you can copy the data files from the master to the new *slave* and then start the slave with *--fastsync*.

Warning: Be careful with *--fastsync*. If the data on both instances is identical, a discrepancy will exist forever.

fastsync is a way to start a slave by starting with an existing master disk image/backup. This option declares that the administrator guarantees the image is correct and completely up-to-date with that of the master. If you have a full and complete copy of data from a master you can use this option to avoid a full synchronization upon starting the slave.

Creating a Slave from an Existing Slave's Disk Image

You can just copy the other *slave's* data file snapshot without any special options. Only take data snapshots when a mongod process is down or locked using db.fsyncLock().

Resyncing a Slave that is too Stale to Recover

Slaves asynchronously apply write operations from the *master* that the slaves poll from the master's *oplog*. The oplog is finite in length, and if a slave is too far behind, a full resync will be necessary. To resync the slave, connect to a slave using the `mongo` and issue the `resync` (page 290) command:

```
use admin
db.runCommand( { resync: 1 } )
```

This forces a full resync of all data (which will be very slow on a large database). You can achieve the same effect by stopping `mongod` on the slave, deleting the entire content of the `dbpath` on the slave, and restarting the `mongod`.

Slave Chaining

Slaves cannot be "chained." They must all connect to the *master* directly.

If a slave attempts "slave from" another slave you will see the following line in the `mongod` long of the shell:

```
assertion 13051 tailable cursor requested on non capped collection ns:local.oplog.$main
```

Correcting a Slave's Source

To change a *slave's* source, manually modify the slave's `local.sources` 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` collection, with the following operation sequence:

```
use local

db.sources.update( { host : "prod.mississippi" }, { $set : { host : "prod.mississippi.example.net" } }
```

Restart the slave with the correct command line arguments or with no *`--source`* option. After configuring `local.sources` 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*.

7.2 Replica Set Tutorials and Procedures

The following tutorials describe a number of common replica set maintenance and operational practices in greater detail.

7.2.1 Getting Started with Replica Sets

Deploy a Replica Set

This tutorial describes how to create a three-member *replica set* from three existing `mongod` instances. The tutorial provides two procedures: one for development and test systems; and a one for production systems.

To instead deploy a replica set from a single standalone MongoDB instance, see *Convert a Standalone to a Replica Set* (page 262). For additional information regarding replica set deployments, see *Replica Set Fundamental Concepts* (page 217) and *Replica Set Architectures and Deployment Patterns* (page 238).

Overview

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

Requirements

Most replica sets consist of three or more `mongod` instances. [15] This tutorial describes a three member set. Production environments should have at least three distinct systems so that each system can run its own instance of `mongod`. For development systems you can run all three instances of the `mongod` process on a local system or within a virtual instance. For production environments, you should maintain as much separation between members as possible. For example, when using virtual machines for production deployments, each member should live on a separate host server, served by redundant power circuits and with redundant network paths.

Procedures

These procedures assume you already have instances of MongoDB installed on the systems you will add as members of your *replica set*. If you have not already installed MongoDB, see the *installation tutorials* (page 3).

Deploy a Development or Test Replica Set The examples in this procedure create a new replica set named `rs0`.

1. Before creating your replica set, verify that every member can successfully connect to every other member. The network configuration must allow all possible connections between any two members. To test connectivity, see *Test Connections Between all Members* (page 234).
2. Start three instances of `mongod` as members of a replica set named `rs0`, as described in this step. For ephemeral tests and the purposes of this guide, you may run the `mongod` instances in separate windows of GNU Screen. OS X and most Linux distributions come with screen installed by default [16] systems.
   (a) Create the necessary data directories by issuing a command similar to the following:

[15] To ensure smooth *elections* (page 218) always design replica sets with odd numbers of members. Use *Arbiters* (page 226) to ensure the set has odd number of voting members and avoid tied elections.

[16] GNU Screen (http://www.gnu.org/software/screen/) is packaged as `screen` on Debian-based, Fedora/Red Hat-based, and Arch Linux.

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

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

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

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

3. Open a `mongo` shell and connect to the first `mongod` instance, with the following command:

```
mongo --port 27017
```

4. Create a replica set configuration object in the `mongo` shell environment to use to initiate the replica set with the following sequence of operations:

```
rsconf = {
           _id: "rs0",
           members: [
                      {
                       _id: 0,
                       host: "<hostname>:27017"
                      }
                     ]
         }
```

5. Use `rs.initiate()` (page 286) to initiate a replica set consisting of the current member and using the default configuration:

```
rs.initiate( rsconf )
```

6. Display the current `replica configuration`:

```
rs.conf()
```

7. Add the second and third `mongod` instances to the replica set using the `rs.add()` (page 287) method. Replace `<hostname>` with your system's hostname in the following examples:

```
rs.add("<hostname>:27018")
rs.add("<hostname>:27019")
```

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

8. Check the status of your replica set at any time with the `rs.status()` (page 286) operation.

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 286)
- `rs.conf()` (page 286)
- `rs.reconfig()` (page 286)
- `rs.add()` (page 287)

You may also consider the simple setup script[17] as an example of a basic automatically configured replica set.

[17] https://github.com/mongodb/mongo-snippets/blob/master/replication/simple-setup.py

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

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

  Configure DNS names appropriately, *or* set up your systems' `/etc/hosts` file to reflect this configuration.
- You specify run-time configuration on each system in a `configuration file` stored in `/etc/mongodb.conf` or in a related location. You *do not* specify run-time configuration through command line options.

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

  ```
  port = 27017

  bind_ip = 10.8.0.10

  dbpath = /srv/mongodb/

  fork = true

  replSet = rs0
  ```

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

  For more documentation on run time options used above and on additional configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

To deploy a production replica set:

1. Before creating your replica set, verify that every member can successfully connect to every other member. The network configuration must allow all possible connections between any two members. To test connectivity, see *Test Connections Between all Members* (page 234).
2. On each system start the `mongod` process by issuing a command similar to following:

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

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

   ```
   mongo
   ```
4. Use `rs.initiate()` (page 286) to initiate a replica set consisting of the current member and using the default configuration:

```
rs.initiate()
```

5. Display the current `replica configuration`:

   ```
   rs.conf()
   ```

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

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

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

7. Check the status of your replica set at any time with the `rs.status()` (page 286) operation.

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 286)
- `rs.conf()` (page 286)
- `rs.reconfig()` (page 286)
- `rs.add()` (page 287)

Convert a Standalone to a Replica Set

This tutorial describes the process for converting a *standalone* `mongod` instance into a three-member *replica set*. Use standalone instances for testing and development, but always use replica sets in production. To install a standalone instance, see the *installation tutorials* (page 3).

To deploy a replica set without using a pre-existing `mongod` instance, see *Deploy a Replica Set* (page 259).

For more information on *replica sets, their use, and administration* (page 217), see:

- *Replica Set Fundamental Concepts* (page 217),
- *Replica Set Architectures and Deployment Patterns* (page 238),
- *Replica Set Operation and Management* (page 223), and
- *Replica Set Considerations and Behaviors for Applications and Development* (page 241).

Note: If you're converting a standalone instance into a replica set that is a *shard* in a *sharded cluster* you must change the shard host information in the *config database*. While connected to a `mongos` instance with a `mongo` shell, 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, and replace `<member,><member,><>` with the list of the members of the replica set.

After completing this operation you must restart all `mongos` instances. When possible you should restart all components of the replica sets (i.e. all `mongos` and all shard `mongod` instances.)

Procedure

This procedure assumes you have a *standalone* instance of MongoDB installed. If you have not already installed MongoDB, see the *installation tutorials* (page 3).

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

   Use a command similar to the following:

   ```
   mongod --port 27017 --dbpath /srv/mongodb/db0 --replSet rs0
   ```

   Replace `/srv/mongodb/db0` with the path of your `dbpath`.

   This starts the instance as a member of a replica set named `rs0`. For more information on configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options` and the `http://docs.mongodb.org/manual/reference/mongod`.

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

   ```
   mongo
   ```

3. Use `rs.initiate()` (page 286) to initiate the replica set:

   ```
   rs.initiate()
   ```

   The set is now operational. To return the replica set configuration, call the `rs.conf()` (page 286) method. To check the status of the replica set, use `rs.status()` (page 286).

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

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

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

   For more information on adding hosts to a replica set, see the *Add Members to a Replica Set* (page 263) document.

Add Members to a Replica Set

Overview

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

Before adding a new member, see the *Adding Members* (page 227) topic in the *Replica Set Operation and Management* (page 223) document.

For background on replication deployment patterns, see the *Replica Set Architectures and Deployment Patterns* (page 238) document.

Requirements

1. An active replica set.

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

If neither of these conditions are satisfied, please use the MongoDB *installation tutorial* (page 3) and the *Deploy a Replica Set* (page 259) tutorial instead.

Procedures

The examples in this procedure use the following configuration:

- The active replica set is `rs0`.
- The new member to be added is `mongodb3.example.net`.
- The `mongod` instance default port is `27017`.
- The `mongodb.conf` configuration file exists in the `/etc` directory and contains the following replica set information:

```
port = 27017

bind_ip = 10.8.0.10

dbpath = /srv/mongodb/db0

logpath = /var/log/mongodb.log

fork = true

replSet = rs0
```

For more information on configuration options, see `http://docs.mongodb.org/manual/reference/configuration-op`

Add a Member to an Existing Replica Set This procedure uses the above *example configuration* (page 264).

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

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

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

     ```
     mongod --dbpath /srv/mongodb/db0 --replSet rs0
     ```

   Replace `/srv/mongodb/db0` with the path of your `dbpath`.

   Take note of the host name and port information for the new `mongod` instance.

2. Open a `mongo` shell connected to the replica set's primary:

   ```
   mongo
   ```

   Note: The primary is the only member that can add or remove members from the replica set. If you do not know which member is the primary, log into any member of the replica set using `mongo` and issue the `db.isMaster()` (page 286) command to determine which member is in the `isMaster.primary` (page 290) field. For example, on the system shell:

   ```
   mongo mongodb0.example.net
   ```

Then in the `mongo` shell:

```
db.isMaster()
```

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

---

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

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

   Note: You can also include the port number, depending on your setup:

   ```
   rs.add("mongodb3.example.net:27017")
   ```

   ---

4. Verify that the member is now part of the replica set by calling the `rs.conf()` (page 286) method, which displays the `replica set configuration`:

   ```
   rs.conf()
   ```

   You can use the `rs.status()` (page 286) function to provide an overview of `replica set status`.

Add a Member to an Existing Replica Set (Alternate Procedure) Alternately, you can add a member to a replica set by specifying an entire configuration document with some or all of the fields in a `members` sub-documents. For example:

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

This configures a *hidden member* that is accessible at `mongodb3.example.net:27017`. See `host`, `priority`, and `hidden` for more information about these settings. When you specify a full configuration object with `rs.add()` (page 287), you must declare the `_id` field, which is not automatically populated in this case.

Production Notes

- In production deployments you likely want to use and configure a *control script* to manage this process based on this command.
- A member can be removed from a set and re-added later. If the removed member's data is still relatively fresh, it can recover and catch up from its old data set. See the `rs.add()` (page 287) and `rs.remove()` (page 289) helpers.
- If you have a backup or snapshot of an existing member, you can move the data files (i.e. `/data/db` or `dbpath`) to a new system and use them to quickly initiate a new member. These files must be:
  - clean: the existing dataset must be from a consistent copy of the database from a member of the same replica set. See the *Backup Strategies for MongoDB Systems* (page 120) document for more information.
  - recent: the copy must more recent than the oldest operation in the *primary* member's *oplog*. The new secondary must be able to become current using operations from the primary's oplog.
- There is a maximum of seven *voting members* (page 251) in any replica set. When adding more members to a replica set that already has seven votes, you must either:
  - add the new member as a *non-voting members* (page 226) or,
  - remove votes from an `existing member`.

Deploy a Geographically Distributed Replica Set

This tutorial outlines the process for deploying a *replica set* with members in multiple locations. The tutorial addresses three-member sets, four-member sets, and sets with more than four members.

For appropriate background, see *Replica Set Fundamental Concepts* (page 217) and *Replica Set Architectures and Deployment Patterns* (page 238). For related tutorials, see *Deploy a Replica Set* (page 259) and *Add Members to a Replica Set* (page 263).

Overview

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

Requirements

For a three-member replica set you need two instances in a primary facility (hereafter, "Site A") and one member in a secondary facility (hereafter, "Site B".) Site A should be the same facility or very close to your primary application infrastructure (i.e. application servers, caching layer, users, etc.)

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

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

- Ensure that a majority of the *voting members* (page 226) are within Site A. This includes *secondary-only members* (page 224) and *arbiters* (page 226) For more information on the need to keep the voting majority on one site, see *Elections and Network Partitions* (page 237).
- If you deploy a replica set with an uneven number of members, deploy an *arbiter* (page 226) on Site A. The arbiter must be on site A to keep the majority there.

For all configurations in this tutorial, deploy each replica set member on a separate system. Although you may deploy more than one replica set member on a single system, doing so reduces the redundancy and capacity of the replica set. Such deployments are typically for testing purposes and beyond the scope of this tutorial.

Procedures

Deploy a Distributed Three-Member Replica Set A geographically distributed three-member deployment has the following features:

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

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

- Ensure that network traffic can pass between all members in the network securely and efficiently. Consider the following:
  - Establish a virtual private network between the systems in Site A and Site B to encrypt all traffic between the sites and remains private. Ensure that your network topology routes all traffic between members within a single site over the local area network.
  - Configure authentication using `auth` and `keyFile`, so that only servers and process with authentication can connect to the replica set.
  - Configure networking and firewall rules so that only traffic (incoming and outgoing packets) on the default MongoDB port (e.g. `27017`) from *within* your deployment.

    See also:

    For more information on security and firewalls, see *Security Considerations for Replica Sets* (page 232).
- Specify run-time configuration on each system in a `configuration file` stored in `/etc/mongodb.conf` or in a related location. *Do not* specify run-time configuration through command line options.

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

  ```
  port = 27017

  bind_ip = 10.8.0.10

  dbpath = /srv/mongodb/

  fork = true

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

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

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

  ```
  replSet = rs0
  ```

  For more documentation on the above run time configurations, as well as additional configuration options, see `http://docs.mongodb.org/manual/reference/configuration-options`.

To deploy a geographically distributed three-member set:

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

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

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

   ---

2. Open a `mongo` shell connected to *one* of the `mongod` instances:

```
mongo
```

3. Use the `rs.initiate()` (page 286) method on *one* member to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

4. Display the current `replica configuration`:

   ```
   rs.conf()
   ```

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

   ```
   rs.add("mongodb1.example.net")
   rs.add("mongodb2.example.net")
   ```

6. Make sure that you have configured the member located in Site B (i.e. `mongodb2.example.net`) as a *secondary-only member* (page 224):

   (a) Issue the following command to determine the `_id` value for `mongodb2.example.net`:

   ```
   rs.conf()
   ```

   (b) In the `members` array, save the `_id` value. The example in the next step assumes this value is `2`.

   (c) In the `mongo` shell connected to the replica set's primary, issue a command sequence similar to the following:

   ```
   cfg = rs.conf()
   cfg.members[2].priority = 0
   rs.reconfig(cfg)
   ```

   Note: In some situations, the `rs.reconfig()` (page 286) shell method can force the current primary to step down and causes an election. When the primary steps down, all clients will disconnect. This is the intended behavior. While, this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods.

   After these commands return you have a geographically distributed three-member replica set.

7. To check the status of your replica set, issue `rs.status()` (page 286).

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 286)
- `rs.conf()` (page 286)
- `rs.reconfig()` (page 286)
- `rs.add()` (page 287)

Deploy a Distributed Four-Member Replica Set A geographically distributed four-member deployment has the following features:

- Each member of the replica set, except for the *arbiter* (see below), resides on its own machine, and the MongoDB processes all bind to port `27017`, which is the standard MongoDB port.
- Each member of the replica set must be accessible by way of resolvable DNS or hostnames in the following scheme:

  - `mongodb0.example.net`
  - `mongodb1.example.net`
  - `mongodb2.example.net`
  - `mongodb3.example.net`

  Configure DNS names appropriately, *or* set up your systems' `/etc/host` file to reflect this configuration. Ensure that one system (e.g. `mongodb2.example.net`) resides in Site B. Host all other systems in Site A.

- One host (e.g. `mongodb3.example.net`) will be an *arbiter* and can run on a system that is also used for an application server or some other shared purpose.
- There are three possible architectures for this replica set:
  - Two members in Site A, two *secondary-only members* (page 224) in Site B, and an arbiter in Site A.
  - Three members in Site A and one secondary-only member in Site B.
  - Two members in Site A, one secondary-only member in Site B, one secondary-only member in Site C, and an arbiter in site A.

  In most cases the first architecture is preferable because it is the least complex.
- Ensure that network traffic can pass between all members in the network securely and efficiently. Consider the following:
  - Establish a virtual private network between the systems in Site A and Site B (and Site C if it exists) to encrypt all traffic between the sites and remains private. Ensure that your network topology routes all traffic between members within a single site over the local area network.
  - Configure authentication using `auth` and `keyFile`, so that only servers and process with authentication can connect to the replica set.
  - Configure networking and firewall rules so that only traffic (incoming and outgoing packets) on the default MongoDB port (e.g. `27017`) from *within* your deployment.

    See also:

    For more information on security and firewalls, see *Security Considerations for Replica Sets* (page 232).
- Specify run-time configuration on each system in a `configuration file` stored in `/etc/mongodb.conf` or in a related location. *Do not* specify run-time configuration through command line options.

  For each MongoDB instance, use the following configuration, with values set appropriate to your systems:

```
port = 27017

bind_ip = 10.8.0.10

dbpath = /srv/mongodb/

fork = true

replSet = rs0/mongodb0.example.net,mongodb1.example.net,mongodb2.example.net,mongodb3.example.ne
```

  Modify `bind_ip` to reflect a secure interface on your system that is able to access all other members of the set *and* that is accessible to all other members of the replica set. The DNS or host names need to point and resolve to this IP address. Configure network rules or a virtual private network (i.e. "VPN") to permit this access.

Note: The portion of the `replSet` following the `http://docs.mongodb.org/manual/` provides a

"seed list" of known members of the replica set. `mongod` uses this list to fetch configuration changes following restarts. It is acceptable to omit this section entirely, and have the `replSet` option resemble:

```
replSet = rs0
```

For more documentation on the above run time configurations, as well as additional configuration options, see doc:*/reference/configuration-options*.

To deploy a geographically distributed four-member set:

1. On each system start the `mongod` process by issuing a command similar to following:

   ```
   mongod --config /etc/mongodb.conf
   ```

   Note: In production deployments you likely want to use and configure a *control script* to manage this process based on this command. Control scripts are beyond the scope of this document.

   ---

2. Open a `mongo` shell connected to this host:

   ```
   mongo
   ```

3. Use `rs.initiate()` (page 286) to initiate a replica set consisting of the current member and using the default configuration:

   ```
   rs.initiate()
   ```

4. Display the current `replica configuration`:

   ```
   rs.conf()
   ```

5. Add the remaining members to the replica set by issuing a sequence of commands similar to the following. The example commands assume the current *primary* is `mongodb0.example.net`:

   ```
   rs.add("mongodb1.example.net")
   rs.add("mongodb2.example.net")
   rs.add("mongodb3.example.net")
   ```

6. In the same shell session, issue the following command to add the arbiter (e.g. `mongodb4.example.net`):

   ```
   rs.addArb("mongodb4.example.net")
   ```

7. Make sure that you have configured each member located in Site B (e.g. `mongodb3.example.net`) as a *secondary-only member* (page 224):

   (a) Issue the following command to determine the `_id` value for the member:

   ```
   rs.conf()
   ```

   (b) In the `members` array, save the `_id` value. The example in the next step assumes this value is `2`.

   (c) In the `mongo` shell connected to the replica set's primary, issue a command sequence similar to the following:

   ```
   cfg = rs.conf()
   cfg.members[2].priority = 0
   rs.reconfig(cfg)
   ```

   Note: In some situations, the `rs.reconfig()` (page 286) shell method can force the current primary to step down and causes an election. When the primary steps down, all clients will disconnect. This is

the intended behavior. While, this typically takes 10-20 seconds, attempt to make these changes during scheduled maintenance periods.

After these commands return you have a geographically distributed four-member replica set.

8. To check the status of your replica set, issue `rs.status()` (page 286).

See also:

The documentation of the following shell functions for more information:

- `rs.initiate()` (page 286)
- `rs.conf()` (page 286)
- `rs.reconfig()` (page 286)
- `rs.add()` (page 287)

Deploy a Distributed Set with More than Four Members The procedure for deploying a geographically distributed set with more than four members is similar to the above procedures, with the following differences:

- Never deploy more than seven voting members.
- Use the procedure for a four-member set if you have an even number of members (see *Deploy a Distributed Four-Member Replica Set* (page 268)). Ensure that Site A always has a majority of the members by deploying the *arbiter* within Site A. For six member sets, deploy at least three voting members in addition to the arbiter in Site A, the remaining members in alternate sites.
- Use the procedure for a three-member set if you have an odd number of members (see *Deploy a Distributed Three-Member Replica Set* (page 266)). Ensure that Site A always has a majority of the members of the set. For example, if a set has five members, deploy three members within the primary facility and two members in other facilities.
- If you have a majority of the members of the set *outside* of Site A and the network partitions to prevent communication between sites, the current primary in Site A will step down, even if none of the members outside of Site A are eligible to become primary.

7.2.2 Replica Set Maintenance and Administration

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 220) is an acceptable size; however, in some situations you may need a larger or smaller oplog. For example, you might need to change the oplog size if your applications perform large numbers of multi-updates or deletes in short periods of time.

This tutorial describes how to resize the oplog. For a detailed explanation of oplog sizing, see the *Oplog* (page 220) topic in the *Replica Set Fundamental Concepts* (page 217) document. For details on the how oplog size affects *delayed members* and affects *replication lag*, see the *Delayed Members* (page 225) topic and the *Check the Replication Lag* (page 233) topic in *Replica Set Operation and Management* (page 223).

Overview

The following is an overview of the procedure for changing the size of the oplog:

1. Shut down the current *primary* instance in the *replica set* and then restart it on a different port and in "standalone" mode.

2. Create a backup of the old (current) oplog. This is optional.
3. Save the last entry from the old oplog.
4. Drop the old oplog.
5. Create a new oplog of a different size.
6. Insert the previously saved last entry from the old oplog into the new oplog.
7. Restart the server as a member of the replica set on its usual port.
8. Apply this procedure to any other member of the replica set that *could become* primary.

Procedure

The examples in this procedure use the following configuration:

- The active *replica set* is `rs0`.
- The replica set is running on port `27017`.
- The replica set is running with a `data directory` of `/srv/mongodb`.

To change the size of the oplog for a replica set, use the following procedure for every member of the set that may become primary.

1. Shut down the `mongod` instance and restart it in "standalone" mode running on a different port.

   Note: Shutting down the *primary* member of the set will trigger a failover situation and another member in the replica set will become primary. In most cases, it is least disruptive to modify the oplogs of all the secondaries before modifying the primary.

   To shut down the current primary instance, use a command that resembles the following:

   ```
   mongod --dbpath /srv/mongodb --shutdown
   ```

   To restart the instance on a different port and in "standalone" mode (i.e. without `replSet` or `--replSet`), use a command that resembles the following:

   ```
   mongod --port 37017 --dbpath /srv/mongodb
   ```

2. Backup the existing oplog on the standalone instance. Use the following sequence of commands:

   ```
   mongodump --db local --collection 'oplog.rs' --port 37017
   ```

   Note: You can restore the backup using the `mongorestore` utility.

   Connect to the instance using the `mongo` shell:

   ```
   mongo --port 37017
   ```

3. Save the last entry from the old (current) oplog.

   (a) In the `mongo` shell, enter the following command to use the `local` database to interact with the oplog:

   ```
   use local
   ```

   (b) Ensure that the temporary collection `temp` is empty by dropping the collection:

```
db.temp.drop()
```

(c) Use the `db.collection.save()` operation to save the last entry in the oplog to a temporary collection:

```
db.temp.save( db.oplog.rs.find( { }, { ts: 1, h: 1 } ).sort( {$natural : -1} ).limit(1).next
```

You can see this oplog entry in the `temp` collection by issuing the following command:

```
db.temp.find()
```

4. Drop the old `oplog.rs` collection in the `local` database. Use the following command:

   ```
   db.oplog.rs.drop()
   ```

   This will return `true` on the shell.

5. Use the `create` command to create a new oplog of a different size. Specify the `size` argument in bytes. A value of `2147483648` will create a new oplog that's 2 gigabytes:

   ```
   db.runCommand( { create : "oplog.rs", capped : true, size : 2147483648 } )
   ```

   Upon success, this command returns the following status:

   ```
   { "ok" : 1 }
   ```

6. Insert the previously saved last entry from the old oplog into the new oplog:

   ```
   db.oplog.rs.save( db.temp.findOne() )
   ```

   To confirm the entry is in the new oplog, issue the following command:

   ```
   db.oplog.rs.find()
   ```

7. Restart the server as a member of the replica set on its usual port:

   ```
   mongod --dbpath /srv/mongodb --shutdown
   mongod --replSet rs0 --dbpath /srv/mongodb
   ```

   The replica member will recover and "catch up" and then will be eligible for election to *primary*. To step down the "temporary" primary that took over when you initially shut down the server, use the `rs.stepDown()` (page 288) method. This will force an election for primary. If the server's *priority* (page 219) is higher than all other members in the set *and* if it has successfully "caught up," then it will likely become primary.

8. Repeat this procedure for all other members of the replica set that are or could become primary.

Force a Member to Become Primary

Synopsis

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 218) as primary. For more information, see *Secondary-Only Members* (page 224).

Procedures

Force a Member to be Primary by Setting its Priority High Changed in version 2.0.

For more information on priorities, see *Member Priority* (page 219).

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 the `mongo` shell, use the following sequence of operations to make `m3.example.net` the primary:

   ```
   cfg = rs.conf()
   cfg.members[0].priority = 0.5
   cfg.members[1].priority = 0.5
   cfg.members[2].priority = 1
   rs.reconfig(cfg)
   ```

   This sets `m3.example.net` to have a higher `local.system.replset.members[n].priority` 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:1000000, force:1})
   ```

This prevents `m1.example.net` from being primary for 1,000,000 seconds, even if there is no other member that can become primary. When `m3.example.net` catches up with `m1.example.net` it will become primary.

If you later want to make `m1.example.net` primary again while it waits for `m3.example.net` to catch up, issue the following command to make `m1.example.net` seek election again:

```
rs.freeze()
```

The `rs.freeze()` (page 289) provides a wrapper around the `replSetFreeze` (page 290) 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()` (page 286) 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()` (page 289) and `rs.stepDown()` (page 288) methods that wrap the `replSetFreeze` (page 290) and `replSetStepDown` commands.

Change Hostnames in a Replica Set

Overview

For most *replica sets* the hostnames [18] in the `host` field never change. However, in some cases you must migrate some or all host names in a replica set as organizational needs change. This document presents two possible procedures for changing the hostnames in the `host` field. Depending on your environments availability requirements, you may:

1. Make the configuration change without disrupting the availability of the replica set. While this ensures that your application will always be able to read and write data to the replica set, this procedure can take a long time and may incur downtime at the application layer. [19]

[18] Always use resolvable hostnames for the value of the `host` field in the replica set configuration to avoid confusion and complexity.

[19] You will have to configure your applications so that they can connect to the replica set at both the old and new locations. This often requires a restart and reconfiguration at the application layer, which may affect the availability of your applications. This re-configuration is beyond the scope of this document and makes the *second option* (page 278) preferable when you must change the hostnames of *all* members of the replica set at once.

For this procedure, see *Changing Hostnames while Maintaining the Replica Set's Availability* (page 277).

2. Stop all members of the replica set at once running on the "old" hostnames or interfaces, make the configuration changes, and then start the members at the new hostnames or interfaces. While the set will be totally unavailable during the operation, the total maintenance window is often shorter.

   For this procedure, see *Changing All Hostnames in Replica Set at Once* (page 278).

See also:

- `http://docs.mongodb.org/manual/reference/replica-configuration`
- *Replica Set Reconfiguration Process*
- `rs.conf()` (page 286) and `rs.reconfig()` (page 286)

And the following tutorials:

- *Deploy a Replica Set* (page 259)
- *Add Members to a Replica Set* (page 263)

Procedures

Given a *replica set* with three members:

- `database0.example.com:27017` (the *primary*)
- `database1.example.com:27017`
- `database2.example.com:27017`

And with the following `rs.conf()` (page 286) output:

```
{
    "_id" : "rs",
    "version" : 3,
    "members" : [
        {
            "_id" : 0,
            "host" : "database0.example.com:27017"
        },
        {
            "_id" : 1,
            "host" : "database1.example.com:27017"
        },
        {
            "_id" : 2,
            "host" : "database2.example.com:27017"
        }
    ]
}
```

The following procedures change the members' hostnames as follows:

- `mongodb0.example.net:27017` (the primary)
- `mongodb1.example.net:27017`
- `mongodb2.example.net:27017`

Use the most appropriate procedure for your deployment.

Changing Hostnames while Maintaining the Replica Set's Availability This procedure uses the above *assumptions* (page 276).

1. For each *secondary* in the replica set, perform the following sequence of operations:
   (a) Stop the secondary.
   (b) Restart the secondary at the new location.
   (c) Open a `mongo` shell connected to the replica set's primary. In our example, the primary runs on port `27017` so you would issue the following command:

   ```
   mongo --port 27017
   ```

   (d) Run the following reconfigure option, for the `host` value where `n` is `1`:

   ```
   cfg = rs.conf()

   cfg.members[1].host = "mongodb1.example.net:27017"

   rs.reconfig(cfg)
   ```

   See `http://docs.mongodb.org/manual/reference/replica-configuration` for more information.
   (e) Make sure your client applications are able to access the set at the new location and that the secondary has a chance to catch up with the other members of the set.

   Repeat the above steps for each non-primary member of the set.
2. Open a `mongo` shell connected to the primary and step down the primary using `replSetStepDown`. In the `mongo` shell, use the `rs.stepDown()` (page 288) wrapper, as follows:

   ```
   rs.stepDown()
   ```

3. When the step down succeeds, shut down the primary.
4. To make the final configuration change, connect to the new primary in the `mongo` shell and reconfigure the `host` value where `n` is `0`:

   ```
   cfg = rs.conf()

   cfg.members[0].host = "mongodb0.example.net:27017"

   rs.reconfig(cfg)
   ```

5. Start the original primary.
6. Open a `mongo` shell connected to the primary.
7. To confirm the new configuration, call `rs.conf()` (page 286) in the `mongo` shell.

   Your output should resemble:

   ```
   {
       "_id" : "rs",
       "version" : 4,
       "members" : [
           {
               "_id" : 0,
               "host" : "mongodb0.example.net:27017"
           },
           {
   ```

```
            "_id" : 1,
            "host" : "mongodb1.example.net:27017"
        },
        {
            "_id" : 2,
            "host" : "mongodb2.example.net:27017"
        }
    ]
}
```

Changing All Hostnames in Replica Set at Once This procedure uses the above *assumptions* (page 276).

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()` (page 286) 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"
        }
    ]
}
```

Convert a Secondary to an Arbiter

If you have a *secondary* in a *replica set* that no longer needs to hold a copy of the data *but* you want to retain it in the set to ensure that the replica set will be able to *elect a primary* (page 218), you can convert the secondary into an *arbiter* (page 226). This document provides two equivalent procedures for this process.

Synopsis

Both of the following procedures are operationally equivalent. Choose whichever procedure you are most comfortable with:

1. You may operate the arbiter on the same port as the former secondary. In this procedure, you must shut down the secondary and remove its data before restarting and reconfiguring it as an arbiter.

   For this procedure, see *Convert a Secondary to an Arbiter and Reuse the Port Number* (page 279).

2. Run the arbiter on a new port. In this procedure, you can reconfigure the server as an arbiter before shutting down the instance running as a secondary.

   For this procedure, see *Convert a Secondary to an Arbiter Running on a New Port Number* (page 280).

See also:

- *Arbiters* (page 226)
- `rs.addArb()` (page 288)
- *Replica Set Operation and Management* (page 223)

Procedures

Convert a Secondary to an Arbiter and Reuse the Port Number

1. If your application is connecting directly to the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Shut down the secondary.

3. Remove the *secondary* from the *replica set* by calling the `rs.remove()` (page 289) 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()` (page 286) 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()` (page 288) method:

   ```
   rs.addArb("<hostname><:port>")
   ```

9. Verify the arbiter belongs to the replica set by calling the `rs.conf()` (page 286) method in the `mongo` shell.

   ```
   rs.conf()
   ```

   The arbiter member should include the following:

   ```
   "arbiterOnly" : true
   ```

Convert a Secondary to an Arbiter Running on a New Port Number

1. If your application is connecting directly to the secondary or has a connection string referencing the secondary, modify the application so that MongoDB queries don't reach the secondary.
2. Create a new, empty data directory to be used with the new port number. For example:

   ```
   mkdir /data/db-temp
   ```

3. Start a new `mongod` instance on the new port number, specifying the new data directory and the existing replica set. Issue a command similar to the following:

   ```
   mongod --port 27021 --dbpath /data/db-temp --replSet rs
   ```

4. In the `mongo` shell connected to the current primary, convert the new `mongod` instance to an arbiter using the `rs.addArb()` (page 288) method:

   ```
   rs.addArb("<hostname><:port>")
   ```

5. Verify the arbiter has been added to the replica set by calling the `rs.conf()` (page 286) 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()` (page 289) 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()` (page 286) 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.

Reconfigure a Replica Set with Unavailable Members

To reconfigure a *replica set* when a **minority** of members are unavailable, use the `rs.reconfig()` (page 286) 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 a **majority** of members are *not* accessible:

- *Reconfigure by Forcing the Reconfiguration* (page 281)
- *Reconfigure by Replacing the Replica Set* (page 282)

You may need to use one of these procedures, for example, in a geographically distributed replica set, where *no* local group of members can reach a majority. See *Elections and Network Partitions* (page 237) for more information on this situation.

Reconfigure by Forcing the Reconfiguration

Changed in version 2.0.

This procedure lets you recover while a majority of *replica set* members are down or unreachable. You connect to any surviving member and use the `force` option to the `rs.reconfig()` (page 286) method.

The `force` option forces a new configuration onto the. Use this procedure only to recover from catastrophic interruptions. Do not use `force` every time you reconfigure. Also, do not use the `force` option in any automatic scripts and do not use `force` when there is still a *primary*.

To force reconfiguration:

1. Back up a surviving member.
2. Connect to a surviving member and save the current configuration. Consider the following example commands for saving the configuration:

```
cfg = rs.conf()

printjson(cfg)
```

3. On the same member, remove the down and unreachable members of the replica set from the `members` array by setting the array equal to the surviving members alone. Consider the following example, which uses the `cfg` variable created in the previous step:

   ```
   cfg.members = [cfg.members[0] , cfg.members[4] , cfg.members[7]]
   ```

4. On the same member, reconfigure the set by using the `rs.reconfig()` (page 286) 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 281).

These procedures are for situations where a *majority* of the *replica set* members are down or unreachable. If a majority is *running*, then skip these procedures and instead use the `rs.reconfig()` (page 286) 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 an invocation that resembles the following:

   ```
   mongod --dbpath /data/db/ --shutdown
   ```

   Set *--dbpath* to the data directory of your `mongod` instance.

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 an invocation that resembles the following:

   ```
   mongod --dbpath /data/db/ --shutdown
   ```

   Set `--dbpath` to the data directory of your `mongod` instance.

2. Move the data directories (i.e. `dbpath`) for all the members except `db0.example.net`, so that all the members except `db0.example.net` have empty data directories. For example:

   ```
   mv /data/db /data/db-old
   ```

3. Move the data files for `local` database (i.e. `local.*`) so that `db0.example.net` has no local database. For example

   ```
   mkdir /data/local-old
   mv /data/db/local* /data/local-old/
   ```

4. Start each member of the replica set normally.

5. Connect to `db0.example.net` in a `mongo` shell and run `rs.initiate()` (page 286) to initiate the replica set.

6. Add the other set members using `rs.add()` (page 287). 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.

Recover MongoDB Data following Unexpected Shutdown

If MongoDB does not shutdown cleanly [20] the on-disk representation of the data files will likely reflect an inconsistent state which could lead to data corruption. [21]

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:

[20] To ensure a clean shut down, use the *`mongod --shutdown`* option, your control script, "Control-C" (when running `mongod` in interactive mode,) or `kill $(pidof mongod)` or `kill -2 $(pidof mongod)`.

[21] 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.

The *Administration* (page 95) documents, including *Replica Set Syncing* (page 252), and the documentation on the `repair`, `repairpath`, and `journal` settings.

Process

Indications When you are aware of a `mongod` instance running without journaling that stops unexpectedly **and** you're not running with replication, you should always run the repair operation before starting MongoDB again. If you're using replication, then restore from a backup and allow replication to perform an initial *sync* (page 252) to restore data.

If the `mongod.lock` file in the data directory specified by `dbpath`, `/data/db` by default, is *not* a zero-byte file, then `mongod` will refuse to start, and you will find a message that contains the following line in your MongoDB log our output:

```
Unclean shutdown detected.
```

This indicates that you need to remove the lockfile and run repair. If you run repair when the `mongodb.lock` file exists without the *`mongod --repairpath`* option, you will see a message that contains the following line:

```
old lock file: /data/db/mongod.lock. probably means unclean shutdown
```

You must remove the lockfile **and** run the repair operation before starting the database normally using the following procedure:

Overview

> **Warning:** Recovering a member of a replica set.
> Do not use this procedure to recover a member of a *replica set*. Instead you should either restore from a *backup* (page 120) or perform an initial sync using data from an intact member of the set, as described in *Resyncing a Member of a Replica Set* (page 231).

There are two processes to repair data files that result from an unexpected shutdown:

1. Use the *`--repair`* option in conjunction with the *`--repairpath`* option. `mongod` will read the existing data files, and write the existing data to new data files. This does not modify or alter the existing data files.

   You do not need to remove the `mongod.lock` file before using this procedure.

2. Use the *`--repair`* option. `mongod` will read the existing data files, write the existing data to new files and replace the existing, possibly corrupt, files with new files.

   You must remove the `mongod.lock` file before using this procedure.

Note: *`--repair`* functionality is also available in the shell with the `db.repairDatabase()` helper for the `repairDatabase` command.

Procedures To repair your data files using the *`--repairpath`* option to preserve the original data files unmodified:

1. Start `mongod` using *`--repair`* to read the existing data files.

   ```
   mongod --dbpath /data/db --repair --repairpath /data/db0
   ```

   When this completes, the new repaired data files will be in the `/data/db0` directory.

2. Start `mongod` using the following invocation to point the `dbpath` at `/data/db0`:

   ```
   mongod --dbpath /data/db0
   ```

Once you confirm that the data files are operational you may delete or archive the data files in the `/data/db` directory.

To repair your data files without preserving the original files, do not use the `--repairpath` option, as in the following procedure:

1. Remove the stale lock file:

   ```
   rm /data/db/mongod.lock
   ```

   Replace `/data/db` with your `dbpath` where your MongoDB instance's data files reside.

   Warning: After you remove the `mongod.lock` file you *must* run the `--repair` process before using your database.

2. Start `mongod` using `--repair` to read the existing data files.

   ```
   mongod --dbpath /data/db --repair
   ```

   When this completes, the repaired data files will replace the original data files in the `/data/db` directory.

3. Start `mongod` using the following invocation to point the `dbpath` at `/data/db`:

   ```
   mongod --dbpath /data/db
   ```

mongod.lock

In normal operation, you should **never** remove the `mongod.lock` file and start `mongod`. Instead consider the one of the above methods to recover the database and remove the lock files. In dire situations you can remove the lockfile, and start the database using the possibly corrupt files, and attempt to recover data from the database; however, it's impossible to predict the state of the database in these situations.

If you are not running with journaling, and your database shuts down unexpectedly for *any* reason, you should always proceed *as if* your database is in an inconsistent and likely corrupt state. If at all possible restore from *backup* (page 120) 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 *Resyncing a Member of a Replica Set* (page 231).

7.3 Replica Set Reference Material

Additionally, consider the following reference listed in this section. The following describes the replica set configuration object:

- `http://docs.mongodb.org/manual/reference/replica-configuration`

The following describe MongoDB output and status related to replication:

- `http://docs.mongodb.org/manual/reference/replica-status`
- `http://docs.mongodb.org/manual/reference/replication-info`

Finally, consider the following quick references of the commands and operations available for replica set administration and use:

7.3.1 Replica Set Commands

This reference collects documentation for all *JavaScript methods* (page 286) for the `mongo` shell that support *replica set* functionality, as well as all *database commands* (page 289) related to replication function.

See *Replication* (page 217), for a list of all replica set documentation.

JavaScript Methods

The following methods apply to replica sets. For a complete list of all methods, see `http://docs.mongodb.org/manual/reference/method`.

`rs.status()`

Returns A *document* with status information.

This output reflects the current status of the replica set, using data derived from the heartbeat packets sent by the other members of the replica set.

This method provides a wrapper around the `replSetGetStatus` (page 291) *database command*.

See also:

"`http://docs.mongodb.org/manual/reference/replica-status`" for documentation of this output.

`db.isMaster()`

Returns a status document with fields that includes the `ismaster` field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* `isMaster` (page 289)

`rs.initiate(`*configuration*`)`

Parameters

- **configuration** – Optional. A *document* that specifies the configuration of a replica set. If not specified, MongoDB will use a default configuration.

Initiates a replica set. Optionally takes a configuration argument in the form of a *document* that holds the configuration of a replica set. Consider the following model of the most basic configuration for a 3-member replica set:

```
{
    _id : <setname>,
     members : [
         {_id : 0, host : <host0>},
         {_id : 1, host : <host1>},
         {_id : 2, host : <host2>},
     ]
}
```

This function provides a wrapper around the "`replSetInitiate` (page 291)" *database command*.

`rs.conf()`

Returns a *document* that contains the current *replica set* configuration object.

`rs.config()`

`rs.config()` (page 286) is an alias of `rs.conf()` (page 286).

`rs.reconfig(`*configuration*`[,` *force* `]`)

Parameters

- **configuration** – A *document* that specifies the configuration of a replica set.
- **force** – Optional. Specify `{ force: true }` as the force parameter to force the replica set to accept the new configuration even if a majority of the members are not accessible. Use with caution, as this can lead to *rollback* situations.

Initializes a new *replica set* configuration. This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.reconfig()` (page 286) provides a wrapper around the "`replSetReconfig` (page 292)" *database command*.

`rs.reconfig()` (page 286) overwrites the existing replica set configuration. Retrieve the current configuration object with `rs.conf()` (page 286), modify the configuration as needed and then use `rs.reconfig()` (page 286) to submit the modified configuration object.

To reconfigure a replica set, use the following sequence of operations:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf)
```

If you want to force the reconfiguration if a majority of the set isn't connected to the current member, or you're issuing the command against a secondary, use the following form:

```
conf = rs.conf()

// modify conf to change configuration

rs.reconfig(conf, { force: true } )
```

> **Warning:** Forcing a `rs.reconfig()` (page 286) can lead to *rollback* situations and other difficult to recover from situations. Exercise caution when using this option.

See also:

"`http://docs.mongodb.org/manual/reference/replica-configuration`" and "*Replica Set Operation and Management* (page 223)".

`rs.add(`*hostspec*, *arbiterOnly*`)`

Specify one of the following forms:

Parameters

- **host** (*string,document*) – Either a string or a document. If a string, specifies a host (and optionally port-number) for a new host member for the replica set; MongoDB will add this host with the default configuration. If a document, specifies any attributes about a member of a replica set.
- **arbiterOnly** (*boolean*) – Optional. If `true`, this host is an arbiter. If the second argument evaluates to `true`, as is the case with some *documents*, then this instance will become an arbiter.

Provides a simple method to add a member to an existing *replica set*. You can specify new hosts in one of two ways:

1. as a "hostname" with an optional port number to use the default configuration as in the *Add a Member to an Existing Replica Set* (page 264) example.
2. as a configuration *document*, as in the *Add a Member to an Existing Replica Set (Alternate Procedure)* (page 265) example.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.add()` (page 287) provides a wrapper around some of the functionality of the "`replSetReconfig` (page 292)" *database command* and the corresponding shell helper `rs.reconfig()` (page 286). See the `http://docs.mongodb.org/manual/reference/replica-configuration` document for full documentation of all replica set configuration options.

Example

To add a `mongod` accessible on the default port `27017` running on the host `mongodb3.example.net`, use the following `rs.add()` (page 287) invocation:

```
rs.add('mongodb3.example.net:27017')
```

If `mongodb3.example.net` is an arbiter, use the following form:

```
rs.add('mongodb3.example.net:27017', true)
```

To add `mongodb3.example.net` as a *secondary-only* (page 224) member of set, use the following form of `rs.add()` (page 287):

```
rs.add( { "_id": 3, "host": "mongodbd3.example.net:27017", "priority": 0 } )
```

Replace, `3` with the next unused `_id` value in the replica set. See `rs.conf()` (page 286) to see the existing `_id` values in the replica set configuration document.

See the `http://docs.mongodb.org/manual/reference/replica-configuration` and *Replica Set Operation and Management* (page 223) documents for more information.

`rs.`**`addArb`**(*hostname*)

Parameters

- **host** (*string*) – Specifies a host (and optionally port-number) for an arbiter member for the replica set.

Adds a new *arbiter* to an existing replica set.

This function will disconnect the shell briefly and forces a reconnection as the replica set renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.`**`stepDown`**(*seconds*)

Parameters

- **seconds** (*init*) – Specify the duration of this operation. If not specified the command uses the default value of 60 seconds.

Returns disconnects shell.

Forces the current replica set member to step down as *primary* and then attempt to avoid election as primary for the designated number of seconds. Produces an error if the current node is not primary.

This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

`rs.stepDown()` (page 288) provides a wrapper around the *database command* `replSetStepDown`.

rs.**freeze**(*seconds*)

Parameters

- **seconds** (*init*) – Specify the duration of this operation.

Forces the current node to become ineligible to become primary for the period specified.

rs.freeze() (page 289) provides a wrapper around the *database command* replSetFreeze (page 290).

rs.**remove**(*hostname*)

Parameters

- **hostname** – Specify one of the existing hosts to remove from the current replica set.

Removes the node described by the hostname parameter from the current *replica set*. This function will disconnect the shell briefly and forces a reconnection as the *replica set* renegotiates which node will be *primary*. As a result, the shell will display an error even if this command succeeds.

Note: Before running the rs.remove() (page 289) operation, you must *shut down* the replica set member that you're removing.

Changed in version 2.2: This procedure is no longer required when using rs.remove() (page 289), but it remains good practice.

---

rs.**slaveOk**()

Provides a shorthand for the following operation:

```
db.getMongo().setSlaveOk()
```

This allows the current connection to allow read operations to run on *secondary* nodes. See the readPref() method for more fine-grained control over *read preference* (page 243) in the mongo shell.

db.**isMaster**()

Returns a status document with fields that includes the ismaster field that reports if the current node is the *primary* node, as well as a report of a subset of current replica set configuration.

This function provides a wrapper around the *database command* isMaster (page 289)

rs.**help**()

Returns a basic help text for all of the *replication* (page 217) related shell functions.

rs.**syncFrom**()

New in version 2.2.

Provides a wrapper around the replSetSyncFrom (page 293), which allows administrators to configure the member of a replica set that the current member will pull data from. Specify the name of the member you want to replicate from in the form of [hostname]:[port].

See replSetSyncFrom (page 293) for more details.

Database Commands

The following commands apply to replica sets. For a complete list of all commands, see http://docs.mongodb.org/manual/reference/commands.

isMaster

The isMaster (page 289) command provides a basic overview of the current replication configuration. MongoDB *drivers* and *clients* use this command to determine what kind of member they're connected to and to discover additional members of a *replica set*. The db.isMaster() (page 286) method provides a wrapper around this database command.

The command takes the following form:

```
{ isMaster: 1 }
```

This command returns a *document* containing the following fields:

isMaster.**setname**
The name of the current replica set, if applicable.

isMaster.**ismaster**
A boolean value that reports when this node is writable. If `true`, then the current node is either a *primary* in a *replica set*, a *master* in a master-slave configuration, or a standalone `mongod`.

isMaster.**secondary**
A boolean value that, when `true`, indicates that the current member is a *secondary* member of a *replica set*.

isMaster.**hosts**
An array of strings in the format of "`[hostname]:[port]`" listing all members of the *replica set* that are not "*hidden*".

isMaster.**arbiter**
An array of strings in the format of "`[hostname]:[port]`" listing all members of the *replica set* that are *arbiters*

Only appears in the `isMaster` (page 289) response for replica sets that have arbiter members.

isMaster.**arbiterOnly**
A boolean value that, when `true` indicates that the current instance is an *arbiter*.

`arbiterOnly` (page 290) only appears in the `isMaster` (page 289) response from arbiters.

isMaster.**primary**
The `[hostname]:[port]` for the current *replica set primary*, if applicable.

isMaster.**me**
The `[hostname]:[port]` of the node responding to this command.

isMaster.**maxBsonObjectSize**
The maximum permitted size of a *BSON* object in bytes for this `mongod` process. If not provided, clients should assume a max size of "`4 * 1024 * 1024`".

isMaster.**localTime**
New in version 2.1.1.

Returns the local server time in UTC. This value is a *ISOdate*. You can use the `toString()` JavaScript method to convert this value to a local date string, as in the following example:

```
db.isMaster().localTime.toString();
```

resync
The `resync` (page 290) command forces an out-of-date slave `mongod` instance to re-synchronize itself. Note that this command is relevant to master-slave replication only. It does not apply to replica sets.

> **Warning:** This command obtains a global write lock and will block other operations until it has completed.

replSetFreeze
The `replSetFreeze` (page 290) command prevents a replica set member from seeking election for the specified number of seconds. Use this command in conjunction with the `replSetStepDown` command to make a different node in the replica set a primary.

The `replSetFreeze` (page 290) command uses the following syntax:

```
{ replSetFreeze: <seconds> }
```

If you want to unfreeze a replica set member before the specified number of seconds has elapsed, you can issue the command with a seconds value of `0`:

```
{ replSetFreeze: 0 }
```

Restarting the `mongod` process also unfreezes a replica set member.

`replSetFreeze` (page 290) is an administrative command, and you must issue the it against the *admin database*.

`replSetGetStatus`

The `replSetGetStatus` command returns the status of the replica set from the point of view of the current server. You must run the command against the *admin database*. The command has the following prototype format:

```
{ replSetGetStatus: 1 }
```

However, you can also run this command from the shell like so:

```
rs.status()
```

See also:

"`http://docs.mongodb.org/manual/reference/replica-status`" and "*Replica Set Fundamental Concepts* (page 217)"

`replSetInitiate`

The `replSetInitiate` (page 291) command initializes a new replica set. Use the following syntax:

```
{ replSetInitiate : <config_document> }
```

The `<config_document>` is a *document* that specifies the replica set's configuration. For instance, here's a config document for creating a simple 3-member replica set:

```
{
    _id : <setname>,
     members : [
         {_id : 0, host : <host0>},
         {_id : 1, host : <host1>},
         {_id : 2, host : <host2>},
     ]
}
```

A typical way of running this command is to assign the config document to a variable and then to pass the document to the `rs.initiate()` (page 286) helper:

```
config = {
    _id : "my_replica_set",
     members : [
         {_id : 0, host : "rs1.example.net:27017"},
         {_id : 1, host : "rs2.example.net:27017"},
         {_id : 2, host : "rs3.example.net", arbiterOnly: true},
     ]
}

rs.initiate(config)
```

```
Notice that omitting the port cause the host to use the default port
```

```
of 27017. Notice also that you can specify other options in the config
documents such as the ``arbiterOnly`` setting in this example.
```

See also:

"http://docs.mongodb.org/manual/reference/replica-configuration," "*Replica Set Operation and Management* (page 223)," and "*Replica Set Reconfiguration*."

replSetMaintenance

The replSetMaintenance (page 292) admin command enables or disables the maintenance mode for a *secondary* member of a *replica set*.

The command has the following prototype form:

```
{ replSetMaintenance: <boolean> }
```

Consider the following behavior when running the replSetMaintenance (page 292) command:

- You cannot run the command on the Primary.
- You must run the command against the admin database.
- When enabled replSetMaintenance: 1, the member enters the RECOVERING state. While the secondary is RECOVERING:
  - The member is not accessible for read operations.
  - The member continues to sync its *oplog* from the Primary.

replSetReconfig

The replSetReconfig (page 292) command modifies the configuration of an existing replica set. You can use this command to add and remove members, and to alter the options set on existing members. Use the following syntax:

```
{ replSetReconfig: <new_config_document>, force: false }
```

You may also run the command using the shell's rs.reconfig() (page 286) method.

Be aware of the following replSetReconfig (page 292) behaviors:

- You must issue this command against the *admin database* of the current primary member of the replica set.
- You can optionally force the replica set to accept the new configuration by specifying force: true. Use this option if the current member is not primary or if a majority of the members of the set are not accessible.

  Warning: Forcing the replSetReconfig (page 292) command can lead to a *rollback* situation. Use with caution.

  Use the force option to restore a replica set to new servers with different hostnames. This works even if the set members already have a copy of the data.
- A majority of the set's members must be operational for the changes to propagate properly.
- This command can cause downtime as the set renegotiates primary-status. Typically this is 10-20 seconds, but could be as long as a minute or more. Therefore, you should attempt to reconfigure only during scheduled maintenance periods.
- In some cases, replSetReconfig (page 292) forces the current primary to step down, initiating an election for primary among the members of the replica set. When this happens, the set will drop all current connections.

Note: replSetReconfig (page 292) obtains a special mutually exclusive lock to prevent more than one replSetReconfig (page 292) operation from occurring at the same time.

replSetSyncFrom

New in version 2.2.

Options

- **host** – Specifies the name and port number of the replica set member that this member replicates from. Use the [hostname]:[port] form.

replSetSyncFrom (page 293) allows you to explicitly configure which host the current mongod will poll *oplog* entries from. This operation may be useful for testing different patterns and in situations where a set member is not replicating from the host you want. The member to replicate from must be a valid source for data in the set.

A member cannot replicate from:

- itself.
- an arbiter, because arbiters do not hold data.
- a member that does not build indexes.
- an unreachable member.
- a mongod instance that is not a member of the same replica set.

If you attempt to replicate from a member that is more than 10 seconds behind the current member, mongod will return and log a warning, but it still *will* replicate from the member that is behind.

If you run rs.syncFrom() (page 289) during initial sync, MongoDB produces no error messages, but the sync target will not change until after the initial sync operation.

The command has the following prototype form:

```
{ replSetSyncFrom: "[hostname]:[port]" }
```

To run the command in the mongo shell, use the following invocation:

```
db.adminCommand( { replSetSyncFrom: "[hostname]:[port]" } )
```

You may also use the rs.syncFrom() (page 289) helper in the mongo shell, in an operation with the following form:

```
rs.syncFrom("[hostname]:[port]")
```

Note: replSetSyncFrom (page 293) and rs.syncFrom() (page 289) provide a temporary override of default behavior. If:

- the mongod instance restarts or
- the connection to the sync target closes;

then, the mongod instance will revert to the default sync logic and target.

7.3.2 Replica Set Features and Version Compatibility

Note: This table is for archival purposes and does not list all features of *replica sets*. Always use the latest stable release of MongoDB in production deployments.

Features	Version
Slave Delay	1.6.3
Hidden	1.7
`replSetFreeze` (page 290) and `replSetStepDown`	1.7.3
Replicated ops in `mongostat`	1.7.3
Syncing from Secondaries	1.8.0
Authentication	1.8.0
Replication from Nearest Server (by ping Time)	2.0
`replSetSyncFrom` (page 293) support for replicating from specific members.	2.2

Additionally:

- 1.8-series secondaries can replicate from 1.6-series primaries.
- 1.6-series secondaries cannot replicate from 1.8-series primaries.

CHAPTER 8

Sharding

Sharding distributes a single logical database system across a cluster of machines. Sharding uses range-based portioning to distribute *documents* based on a specific *shard key*.

For a general introduction to sharding, cluster operations, and relevant implications and administration see: *FAQ: Sharding with MongoDB* (page 373).

8.1 Sharded Cluster Use and Operation

The documents in this section introduce sharded clusters, their operation, functioning, and use. If you are unfamiliar with data partitioning, or MongoDB's sharding implementation begin with these documents:

8.1.1 Sharded Cluster Overview

Sharding is MongoDB's approach to scaling out. Sharding partitions a collection and stores the different portions on different machines. When a database's collections become too large for existing storage, you need only add a new machine. Sharding automatically distributes collection data to the new server.

Sharding automatically balances data and load across machines. Sharding provides additional write capacity by distributing the write load over a number of `mongod` instances. Sharding allows users to increase the potential amount of data in the *working set*.

How Sharding Works

To run sharding, you set up a sharded cluster. For a description of sharded clusters, see *Sharded Cluster Administration* (page 298).

Within a sharded cluster, you enable sharding on a per-database basis. After enabling sharding for a database, you choose which collections to shard. For each sharded collection, you specify a *shard key*.

The shard key determines the distribution of the collection's *documents* among the cluster's *shards*. The shard key is a *field* that exists in every document in the collection. MongoDB distributes documents according to ranges of values in the shard key. A given shard holds documents for which the shard key falls within a specific range of values. Shard keys, like *indexes*, can be either a single field or multiple fields.

Within a shard, MongoDB further partitions documents into *chunks*. Each chunk represents a smaller range of values within the shard's range. When a chunk grows beyond the *chunk size* (page 309), MongoDB *splits* the chunk into smaller chunks, always based on ranges in the shard key.

Shard Key Selection

Choosing the correct shard key can have a great impact on the performance, capability, and functioning of your database and cluster. Appropriate shard key choice depends on the schema of your data and the way that your application queries and writes data to the database.

The ideal shard key:

- is easily divisible which makes it easy for MongoDB to distribute content among the shards. Shard keys that have a limited number of possible values are not ideal as they can result in some chunks that are "unsplitable." See the *Cardinality* (page 305) section for more information.
- will distribute write operations among the cluster, to prevent any single shard from becoming a bottleneck. Shard keys that have a high correlation with insert time are poor choices for this reason; however, shard keys that have higher "randomness" satisfy this requirement better. See the *Write Scaling* (page 305) section for additional background.
- will make it possible for the `mongos` to return most query operations directly from a single *specific* `mongod` instance. Your shard key should be the primary field used by your queries, and fields with a high degree of "randomness" are poor choices for this reason. See the *Query Isolation* (page 306) section for specific examples.

The challenge when selecting a shard key is that there is not always an obvious choice. Often, an existing field in your collection may not be the optimal key. In those situations, computing a special purpose shard key into an additional field or using a compound shard key may help produce one that is more ideal.

Important: Shard keys are immutable and cannot be changed after insertion.

Shard Balancing

Balancing is the process MongoDB uses to redistribute data within a *sharded cluster*. When a *shard* has too many *chunks* when compared to other shards, MongoDB automatically balances the shards. MongoDB balances the shards without intervention from the application layer.

The balancing process attempts to minimize the impact that balancing can have on the cluster, by:

- Moving only one chunk at a time.
- Initiating a balancing round **only** when the difference in the number of chunks between the shard with the greatest number and the shard with the lowest exceeds the *migration threshold* (page 308).

You may disable the balancer on a temporary basis for maintenance and limit the window during which it runs to prevent the balancing process from impacting production traffic.

See also:

Manage Sharded Cluster Balancer (page 326) and *Sharded Cluster Internals and Behaviors* (page 304).

Note: The balancing procedure for *sharded clusters* is entirely transparent to the user and application layer. This documentation is only included for your edification and possible troubleshooting purposes.

When to Use Sharding

While sharding is a powerful and compelling feature, it comes with significant *Infrastructure Requirements for Sharded Clusters* (page 297) and some limited complexity costs. As a result, use sharding only as necessary, and when indicated by actual operational requirements. Consider the following overview of indications it may be time to consider sharding.

You should consider deploying a *sharded cluster*, if:

- your data set approaches or exceeds the storage capacity of a single node in your system.
- the size of your system's active *working set will soon* exceed the capacity of the *maximum* amount of RAM for your system.
- your system has a large amount of write activity, a single MongoDB instance cannot write data fast enough to meet demand, and all other approaches have not reduced contention.

If these attributes are not present in your system, sharding will only add additional complexity to your system without providing much benefit. When designing your data model, if you will eventually need a sharded cluster, consider which collections you will want to shard and the corresponding shard keys.

Warning: It takes time and resources to deploy sharding, and if your system has *already* reached or exceeded its capacity, you will have a difficult time deploying sharding without impacting your application.
As a result, if you think you will need to partition your database in the future, **do not** wait until your system is overcapacity to enable sharding.

Infrastructure Requirements for Sharded Clusters

A *sharded cluster* has the following components:

- Three *config servers*.

  These special `mongod` instances store the metadata for the cluster. The `mongos` instances cache this data and use it to determine which *shard* is responsible for which *chunk*.

  For development and testing purposes you may deploy a cluster with a single configuration server process, but always use exactly three config servers for redundancy and safety in production.

- Two or more shards. Each shard consists of one or more `mongod` instances that store the data for the shard.

  These "normal" `mongod` instances hold all of the actual data for the cluster.

  Typically each shard is a *replica sets*. Each replica set consists of multiple `mongod` instances. The members of the replica set provide redundancy and high available for the data in each shard.

  Warning: MongoDB enables data *partitioning*, or sharding, on a *per collection* basis. You *must* access all data in a sharded cluster via the `mongos` instances as below. If you connect directly to a `mongod` in a sharded cluster you will see its fraction of the cluster's data. The data on any given shard may be somewhat random: MongoDB provides no guarantee that any two contiguous chunks will reside on a single shard.

- One or more `mongos` instances.

  These instance direct queries from the application layer to the shards that hold the data. The `mongos` instances have no persistent state or data files and only cache metadata in RAM from the config servers.

  Note: In most situations `mongos` instances use minimal resources, and you can run them on your application servers without impacting application performance. However, if you use the *aggregation framework* some processing may occur on the `mongos` instances, causing that `mongos` to require more system resources.

Data Quantity Requirements for Sharded Clusters

Your cluster must manage a significant quantity of data for sharding to have an effect on your collection. The default *chunk* size is 64 megabytes, and the *balancer* (page 296) will not begin moving data until the imbalance of chunks in the cluster exceeds the *migration threshold* (page 308).

Practically, this means that unless your cluster has many hundreds of megabytes of data, chunks will remain on a single shard.

While there are some exceptional situations where you may need to shard a small collection of data, most of the time the additional complexity added by sharding the small collection is not worth the additional complexity and overhead unless you need additional concurrency or capacity for some reason. If you have a small data set, usually a properly configured single MongoDB instance or replica set will be more than sufficient for your persistence layer needs.

Chunk size is `user configurable`. However, the default value is of 64 megabytes is ideal for most deployments. See the *Chunk Size* (page 309) section in the *Sharded Cluster Internals and Behaviors* (page 304) document for more information.

8.1.2 Sharded Cluster Administration

Sharding occurs within a *sharded cluster*. A sharded cluster consists of the following components:

- *Shards* (page 298). Each shard is a separate `mongod` instance or *replica set* that holds a portion of the database collections.
- *Config servers* (page 298). Each config server is a `mongod` instance that holds metadata about the cluster. The metadata maps *chunks* to shards.
- *mongos instances* (page 299). The `mongos` instances route the reads and writes to the shards.

See also:

- For specific configurations, see *Sharded Cluster Architectures* (page 302).
- To set up sharded clusters, see *Deploy a Sharded Cluster* (page 312).

Shards

A shard is a container 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 should be replica sets.

Applications do not access the shards directly. Instead, the *mongos instances* (page 299) routes reads and writes from applications to the shards.

Config Servers

Config servers maintain the shard metadata in a config database. The *config database* stores the relationship between *chunks* and where they reside within a *sharded cluster*. Without a config database, the `mongos` instances would be unable to route queries or write operations within the cluster.

Config servers *do not* run as replica sets. Instead, a *cluster* operates with a group of *three* config servers that use a two-phase commit process that ensures immediate consistency and reliability.

For testing purposes you may deploy a cluster with a single config server, but this is not recommended for production.

> **Warning:** If your cluster has a single config server, this `mongod` is a single point of failure. If the instance is inaccessible the cluster is not accessible. If you cannot recover the data on a config server, the cluster will be inoperable.
> **Always** use three config servers for production deployments.

The actual load on configuration servers is small because each `mongos` instance maintains a cached copy of the configuration database. MongoDB only writes data to the config server to:

- create splits in existing chunks, which happens as data in existing chunks exceeds the maximum chunk size.
- migrate a chunk between shards.

Additionally, all config servers must be available on initial setup of a sharded cluster, each `mongos` instance must be able to write to the `config.version` collection.

If one or two configuration instances become unavailable, the cluster's metadata becomes *read only*. It is still possible to read and write data from the shards, but no chunk migrations or splits will occur until all three servers are accessible. At the same time, config server data is only read in the following situations:

- A new `mongos` starts for the first time, or an existing `mongos` restarts.
- After a chunk migration, the `mongos` instances update themselves with the new cluster metadata.

If all three config servers are inaccessible, you can continue to use the cluster as long as you don't restart the `mongos` instances until after config servers are accessible again. If you restart the `mongos` instances and there are no accessible config servers, the `mongos` would be unable to direct queries or write operations to the cluster.

Because the configuration data is small relative to the amount of data stored in a cluster, the amount of activity is relatively low, and 100% up time is not required for a functioning sharded cluster. As a result, backing up the config servers is not difficult. Backups of config servers are critical as clusters become totally inoperable when you lose all configuration instances and data. Precautions to ensure that the config servers remain available and intact are critical.

Note: Configuration servers store metadata for a single sharded cluster. You must have a separate configuration server or servers for each cluster you administer.

Sharded Cluster Operations and `mongos` Instances

The `mongos` program provides a single unified interface to a sharded cluster for applications using MongoDB. Except for the selection of a *shard key*, application developers and administrators need not consider any of the *internal details of sharding* (page 304).

`mongos` caches data from the *config server* (page 298), and uses this to route operations from applications and clients to the `mongod` instances. `mongos` have no *persistent* state and consume 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.

Automatic Operation and Query Routing with `mongos`

`mongos` uses information from *config servers* (page 298) to route operations to the cluster as efficiently as possible. In general, operations in a sharded environment are either:

1. Targeted at a single shard or a limited group of shards based on the shard key.
2. Broadcast to all shards in the cluster that hold documents in a collection.

When possible you should design your operations to be as targeted as possible. Operations have the following targeting characteristics:

- Query operations broadcast to all shards [1] **unless** the `mongos` can determine which shard or shard stores this data.

  For queries that include the shard key, `mongos` can target the query at a specific shard or set of shards, if the portion of the shard key included in the query is a *prefix* of the shard key. For example, if the shard key is:

  ```
  { a: 1, b: 1, c: 1 }
  ```

  The `mongos` program *can* route queries that include the full shard key or either of the following shard key prefixes at a specific shard or set of shards:

  ```
  { a: 1 }
  { a: 1, b: 1 }
  ```

  Depending on the distribution of data in the cluster and the selectivity of the query, `mongos` may still have to contact multiple shards [2] to fulfill these queries.

- All `insert()` operations target to one shard.
- All single `update()` operations target to one shard. This includes *upsert* operations.
- The `mongos` broadcasts multi-update operations to every shard.
- The `mongos` broadcasts `remove()` operations to every shard unless the operation specifies the shard key in full.

While some operations must broadcast to all shards, you can improve performance by using as many targeted operations as possible by ensuring that your operations include the shard key.

Sharded Query Response Process

To route a query to a *cluster*, `mongos` uses the following process:

1. Determine the list of *shards* that must receive the query.

   In some cases, when the *shard key* or a prefix of the shard key is a part of the query, the `mongos` can route the query to a subset of the shards. Otherwise, the `mongos` must direct the query to *all* shards that hold documents for that collection.

   Example

   Given the following shard key:

   ```
   { zipcode: 1, u_id: 1, c_date: 1 }
   ```

   Depending on the distribution of chunks in the cluster, the `mongos` may be able to target the query at a subset of shards, if the query contains the following fields:

   ```
   { zipcode: 1 }
   { zipcode: 1, u_id: 1 }
   { zipcode: 1, u_id: 1, c_date: 1 }
   ```

   ---

2. Establish a cursor on all targeted shards.

   When the first batch of results returns from the cursors:

   (a) For query with sorted results (i.e. using `cursor.sort()`) the `mongos` instance performs a merge sort of all queries.

[1] If a shard does not store chunks from a given collection, queries for documents in that collection are not broadcast to that shard.
[2] `mongos` will route some queries, even some that include the shard key, to all shards, if needed.

(b) For a query with unsorted results, the `mongos` instance returns a result cursor that "round robins" results from all cursors on the shards.

Changed in version 2.0.5: Before 2.0.5, the `mongos` exhausted each cursor, one by one.

Sharded Cluster Security Considerations

MongoDB controls access to *sharded clusters* with key files that store authentication credentials. The components of sharded clusters use the secret stored in the key files when authenticating to each other. Create key files and then point your `mongos` and `mongod` instances to the files, as described later in this section.

Beyond the `auth` mechanisms described in this section, always run your sharded clusters in trusted networking environments that limit access to the cluster with network rules. Your networking environments should enforce restrictions that ensure only known traffic reaches your `mongos` and `mongod` instances.

This section describes authentication specific to sharded clusters. For information on authentication across MongoDB, see *Authentication* (page 136).

Access Control Privileges in Sharded Clusters

In sharded clusters, MongoDB provides separate administrative privileges for the sharded cluster and for each shard. Beyond these administration privileges, privileges for sharded cluster deployments are functionally the same as any other MongoDB deployment. See, *Authentication* (page 136) for more information.

For sharded clusters, MongoDB provides these separate administrative privileges:

- Administrative privileges for the sharded cluster. These privileges provide read-and-write access to the config servers' '*admin*. These users can run all administrative commands. Administrative privileges also give the user read-and-write access to all the cluster's databases.

  The credentials for administrative privileges on the cluster reside on the config servers. To receive admin access to the cluster, you must authenticate a session while connected to a `mongos` instance using the admin database.

- Administrative privileges for the `mongod` instance, or *replica set*, that provides each individual shard. Each shard has its own admin database that stores administrative credentials and access for that shard only. These credentials are *completely* distinct from the cluster-wide administrative credentials.

  As with all `mongod` instances, MongoDB provides two types of administrative privileges for a shard:

  – Normal administrative privileges, which provide read-and-write access to the admin database and access to all administrative commands, and which provide read-and-write access to all other databases on that shard.

  – Read-only administrative privileges, which provide read-only access to the admin database and to all other databases on that shard.

Also, as with all `mongod` instances, a MongoDB sharded cluster provides the following non-administrative user privileges:

- Normal privileges, which provide read-and-write access to a specific database. Users with normal privilege can add users to the database.
- Read-only privileges, which provide read-only access to a specific database.

For more information on privileges, see *Authentication* (page 136).

Enable Authentication in a Sharded Cluster

New in version 2.0: Support for authentication with sharded clusters.

To control access to a sharded cluster, create key files and then set the `keyFile` option on *all* components of the sharded cluster, including all `mongos` instances, all config server `mongod` instances, and all shard `mongod` instances. The content of the key file is arbitrary but must be the same on all cluster members.

To enable authentication, do the following:

1. Generate a key file to store authentication information, as described in the *Generate a Key File* (page 148) section.
2. On each component in the sharded cluster, enable authentication by doing one of the following:
   - In the configuration file, set the `keyFile` option to the key file's path and then start the component, as in the following example:

     ```
     keyFile = /srv/mongodb/keyfile
     ```

   - When starting the component, set *`--keyFile`* option, which is an option for both `mongos` instances and `mongod` instances. Set the *`--keyFile`* to the key file's path.

   Note: The `keyFile` setting implies `auth`, which means in most cases you do not need to set `auth` explicitly.
3. Add the first administrative user and then add subsequent users. See *Add Users* (page 146).

Access a Sharded Cluster with Authentication

To access a sharded cluster as an authenticated admin user, see *Administrative Access in MongoDB* (page 147).

To access a sharded cluster as an authenticated, non-admin user, see either of the following:

- `authenticate`
- `db.auth()`

To terminate an authenticated session, see the `logout` command.

8.1.3 Sharded Cluster Architectures

This document describes the organization and design of *sharded cluster* deployments.

Restriction on the Use of the `localhost` Interface

Because all components of a *sharded cluster* must communicate with each other over the network, there are special restrictions regarding the use of localhost addresses:

If you use either "localhost" or "`127.0.0.1`" as the host identifier, then you must use "localhost" or "`127.0.0.1`" for *all* host settings for any MongoDB instances in the cluster. This applies to both the `host` argument to `addShard` (page 351) and the value to the *`mongos --configdb`* run time option. If you mix localhost addresses with remote host address, MongoDB will produce errors.

Test Cluster Architecture

You can deploy a very minimal cluster for testing and development. These *non-production* clusters have the following components:

- One *config server* (page 298).
- At least one `mongod` instance (either *replica sets* or as a standalone node.)
- One `mongos` instance.

Warning: Use the test cluster architecture for testing and development only.

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-level cluster must have the following components:

- Three *config servers* (page 298), each residing on a discrete system.

  A single *sharded cluster* must have exclusive use of its *config servers* (page 298). If you have multiple sharded clusters, you will need to have a group of config servers for each cluster.

- Two or more *replica sets* to serve as *shards*. For information on replica sets, see *Replication* (page 217).
- Two or more `mongos` instances. Typically, you deploy a single `mongos` instance on each application server. Alternatively, you may deploy several `mongos` nodes and let your application connect to these via a load balancer.

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. [3] However, in production deployments some databases and collections will use sharding, while other databases and collections will only reside on a single database instance or replica set (i.e. a *shard*.)

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.

Every database has a "primary" [4] shard that holds all un-sharded collections in that database. All collections that *are not* sharded reside on the primary for their database. Use the `movePrimary` command to change the primary shard for a database. Use the `db.printShardingStatus()` command or the `sh.status()` (page 349) to see an overview of the cluster, which contains information about the *chunk* and database distribution within the cluster.

Warning: The `movePrimary` command can be expensive because it copies all non-sharded data to the new shard, during which that data will be unavailable for other operations.

When you deploy a new *sharded cluster*, the "first shard" becomes the primary for all databases before enabling sharding. Databases created subsequently, may reside on any shard in the cluster.

[3] As you configure sharding, you will use the `enableSharding` (page 352) command to enable sharding for a database. This simply makes it possible to use the `shardCollection` (page 352) command on a collection within that database.

[4] The term "primary" in the context of databases and sharding, has nothing to do with the term *primary* in the context of *replica sets*.

High Availability and MongoDB

A *production* (page 303) *cluster* has no single point of failure. This section introduces the availability concerns for MongoDB deployments and highlights potential failure scenarios and available resolutions:

- Application servers or `mongos` instances become unavailable.

  If each application server has its own `mongos` instance, other application servers can continue access the database. Furthermore, `mongos` instances do not maintain persistent state, and they can restart and become unavailable without loosing any state or data. When a `mongos` instance starts, it retrieves a copy of the *config database* and can begin routing queries.

- A single `mongod` becomes unavailable in a shard.

  Replica sets (page 217) provide high availability for shards. If the unavailable `mongod` is a *primary*, then the replica set will *elect* (page 218) 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. [5]

  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 database* become unavailable.

  Three distinct `mongod` instances provide the *config database* using a special two-phase commits to maintain consistent state between these `mongod` instances. Cluster operation will continue as normal but *chunk migration* (page 296) and the cluster can create no new *chunk splits* (page 320). Replace the config server as soon as possible. If all multiple config databases become unavailable, the cluster can become inoperable.

  Note: All config servers must be running and available when you first initiate a *sharded cluster*.

8.1.4 Sharded Cluster Internals and Behaviors

This document introduces lower level sharding concepts for users who are familiar with *sharding* generally and want to learn more about the internals. This document provides a more detailed understanding of your cluster's behavior. For higher level sharding concepts, see *Sharded Cluster Overview* (page 295). For complete documentation of sharded clusters see the *Sharding* (page 295) section of this manual.

Shard Keys

Shard keys are the field in a collection that MongoDB uses to distribute *documents* within a sharded cluster. See the *overview of shard keys* (page 296) for an introduction to these topics.

[5] 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*.

Cardinality

Cardinality in the context of MongoDB, refers to the ability of the system to *partition* data into *chunks*. For example, consider a collection of data such as an "address book" that stores address records:

- Consider the use of a `state` field as a shard key:

  The state key's value holds the US state for a given address document. This field has a *low cardinality* as all documents that have the same value in the `state` field *must* reside on the same shard, even if a particular state's chunk exceeds the maximum chunk size.

  Since there are a limited number of possible values for the `state` field, MongoDB may distribute data unevenly among a small number of fixed chunks. This may have a number of effects:

  - If MongoDB cannot split a chunk because all of its documents have the same shard key, migrations involving these un-splittable chunks will take longer than other migrations, and it will be more difficult for your data to stay balanced.
  - If you have a fixed maximum number of chunks, you will never be able to use more than that number of shards for this collection.

- Consider the use of a `zipcode` field as a shard key:

  While this field has a large number of possible values, and thus has potentially higher cardinality, it's possible that a large number of users could have the same value for the shard key, which would make this chunk of users un-splittable.

  In these cases, cardinality depends on the data. If your address book stores records for a geographically distributed contact list (e.g. "Dry cleaning businesses in America,") then a value like zipcode would be sufficient. However, if your address book is more geographically concentrated (e.g "ice cream stores in Boston Massachusetts,") then you may have a much lower cardinality.

- Consider the use of a `phone-number` field as a shard key:

  Phone number has a *high cardinality,* because users will generally have a unique value for this field, MongoDB will be able to split as many chunks as needed.

While "high cardinality," is necessary for ensuring an even distribution of data, having a high cardinality does not guarantee sufficient *query isolation* (page 306) or appropriate *write scaling* (page 305). Please continue reading for more information on these topics.

Write Scaling

Some possible shard keys will allow your application to take advantage of the increased write capacity that the cluster can provide, while others do not. Consider the following example where you shard by the values of the default *_id* field, which is *ObjectId*.

`ObjectId` is computed upon document creation, that is a unique identifier for the object. However, the most significant bits of data in this value represent a time stamp, which means that they increment in a regular and predictable pattern. Even though this value has *high cardinality* (page 305), 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 306), which is another important characteristic of shard keys.

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 298), 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 *mongos and Sharding* (page 299) and *config server* (page 298) sections for a more general overview of querying in sharded environments.

Query Isolation The fastest queries in a sharded environment are those that `mongos` will route to a single shard, using the *shard key* and the cluster meta data from the *config server* (page 298). For queries that don't include the shard key, `mongos` must query all shards, wait for their response and then return the result to the application. These "scatter/gather" queries can be long running operations.

If your query includes the first component of a compound shard key [6], the `mongos` can route the query directly to a single shard, or a small number of shards, which provides better performance. Even if you query values of the shard key reside in different chunks, the `mongos` will route queries directly to specific shards.

To select a shard key for a collection:

- determine the most commonly included fields in queries for a given application
- find which of these operations are most performance dependent.

If this field has low cardinality (i.e not sufficiently selective) you should add a second field to the shard key making a compound shard key. The data may become more splittable with a compound shard key.

See

Sharded Cluster Operations and mongos Instances (page 299) for more information on query operations in the context of sharded clusters. Specifically the *Automatic Operation and Query Routing with mongos* (page 299) sub-section outlines the procedure that `mongos` uses to route read operations to the shards.

Sorting In sharded systems, the `mongos` performs a merge-sort of all sorted query results from the shards. See the *sharded query routing* (page 299) and *Use Indexes to Sort Query Results* (page 201) sections for more information.

Operations and Reliability

The most important consideration when choosing a *shard key* are:

- to ensure that MongoDB will be able to distribute data evenly among shards, and
- to scale writes across the cluster, and
- to ensure that `mongos` can isolate most queries to a specific `mongod`.

Furthermore:

[6] In many ways, you can think of the shard key a cluster-wide unique index. However, be aware that sharded systems cannot enforce cluster-wide unique indexes *unless* the unique field is in the shard key. Consider the *Indexing Overview* (page 185) page for more information on indexes and compound indexes.

- 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.

Choosing a Shard Key

It is unlikely that any single, naturally occurring key in your collection will satisfy all requirements of a good shard key. There are three options:

1. Compute a more ideal shard key in your application layer, and store this in all of your documents, potentially in the `_id` field.
2. Use a compound shard key, that uses two or three values from all documents that provide the right mix of cardinality with scalable write operations and query isolation.
3. Determine that the impact of using a less than ideal shard key, is insignificant in your use case given:
   - limited write volume,
   - expected data size, or
   - query patterns and demands.

From a decision making stand point, begin by finding the field that will provide the required *query isolation* (page 306), ensure that *writes will scale across the cluster* (page 306), and then add an additional field to provide additional *cardinality* (page 305) if your primary key does not have sufficient split-ability.

Shard Key Indexes

All sharded collections **must** have an index that starts with the *shard key*. If you shard a collection that does not yet contain documents and *without* such an index, the `shardCollection` (page 352) command will create an index on the shard key. If the collection already contains documents, you must create an appropriate index before using `shardCollection` (page 352).

Changed in version 2.2: The index on the shard key no longer needs to be identical to the shard key. This index can be an index of the shard key itself as before, or a *compound index* where the shard key is the prefix of the index. This index *cannot* be a multikey index.

If you have a collection named `people`, sharded using the field `{ zipcode: 1 }`, and you want to replace this with an index on the field `{ zipcode: 1, username: 1 }`, then:

1. Create an index on `{ zipcode: 1, username: 1 }`:

   ```
   db.people.ensureIndex( { zipcode: 1, username: 1 } );
   ```

2. When MongoDB finishes building the index, you can safely drop existing index on `{ zipcode: 1 }`:

   ```
   db.people.dropIndex( { zipcode: 1 } );
   ```

Warning: The index on the shard key **cannot** be a multikey index.
As above, an index on `{ zipcode: 1, username: 1 }` can only replace an index on `zipcode` if there are no array values for the `username` field.
If you drop the last appropriate index for the shard key, recover by recreating an index on just the shard key.

Cluster Balancer

The *balancer* (page 296) sub-process is responsible for redistributing chunks evenly among the shards and ensuring that each member of the cluster is responsible for the same volume of data. This section contains complete documentation of the balancer process and operations. For a higher level introduction see the *Shard Balancing* (page 296) section.

Balancing Internals

A balancing round originates from an arbitrary `mongos` instance from one of the cluster's `mongos` instances. When a balancer process is active, the responsible `mongos` acquires a "lock" by modifying a document in the `lock` collection in the *config-database*.

By default, the balancer process is always running. When the number of chunks in a collection is unevenly distributed among the shards, the balancer begins migrating *chunks* from shards with more chunks to shards with a fewer number of chunks. The balancer will continue migrating chunks, one at a time, until the data is evenly distributed among the shards.

While these automatic chunk migrations are crucial for distributing data, they carry some overhead in terms of bandwidth and workload, both of which can impact database performance. As a result, MongoDB attempts to minimize the effect of balancing by only migrating chunks when the distribution of chunks passes the *migration thresholds* (page 308).

The migration process ensures consistency and maximizes availability of chunks during balancing: when MongoDB begins migrating a chunk, the database begins copying the data to the new server and tracks incoming write operations. After migrating chunks, the "from" `mongod` sends all new writes to the "receiving" server. Finally, `mongos` updates the chunk record in the *config database* to reflect the new location of the chunk.

Note: Changed in version 2.0: Before MongoDB version 2.0, large differences in timekeeping (i.e. clock skew) between `mongos` instances could lead to failed distributed locks, which carries the possibility of data loss, particularly with skews larger than 5 minutes. Always use the network time protocol (NTP) by running `ntpd` on your servers to minimize clock skew.

Migration Thresholds

Changed in version 2.2: The following thresholds appear first in 2.2; prior to this release, balancing would only commence if the shard with the most chunks had 8 more chunks than the shard with the least number of chunks.

In order to minimize the impact of balancing on the cluster, the *balancer* will not begin balancing until the distribution of chunks has reached certain thresholds. These thresholds apply to the difference in number of *chunks* between the shard with the greatest number of chunks and the shard with the least number of chunks. The balancer has the following thresholds:

Number of Chunks	Migration Threshold
Less than 20	2
20-79	4
80 and greater	8

Once a balancing round starts, the balancer will not stop until the difference between the number of chunks on any two shards is *less than two* or a chunk migration fails.

Note: You can restrict the balancer so that it only operates between specific start and end times. See *Schedule the Balancing Window* (page 327) for more information.

The specification of the balancing window is relative to the local time zone of all individual `mongos` instances in the sharded cluster.

Chunk Size

The default *chunk* size in MongoDB is 64 megabytes.

When chunks grow beyond the *specified chunk size* (page 309) a `mongos` instance will split the chunk in half. This will eventually lead to migrations, when chunks become unevenly distributed among the cluster. The `mongos` instances will initiate a round of migrations to redistribute data in the cluster.

Chunk size is arbitrary and must account for the following:

1. Small chunks lead to a more even distribution of data at the expense of more frequent migrations, which creates expense at the query routing (`mongos`) layer.
2. Large chunks lead to fewer migrations, which is more efficient both from the networking perspective *and* in terms internal overhead at the query routing layer. Large chunks produce these efficiencies at the expense of a potentially more uneven distribution of data.

For many deployments it makes sense to avoid frequent and potentially spurious migrations at the expense of a slightly less evenly distributed data set, but this value is *configurable* (page 322). Be aware of the following limitations when modifying chunk size:

- Automatic splitting only occurs when inserting *documents* or updating existing documents; if you lower the chunk size it may take time for all chunks to split to the new size.
- Splits cannot be "undone:" if you increase the chunk size, existing chunks must grow through insertion or updates until they reach the new size.

Note: Chunk ranges are inclusive of the lower boundary and exclusive of the upper boundary.

Shard Size

By default, MongoDB will attempt to fill all available disk space with data on every shard as the data set grows. Monitor disk utilization in addition to other performance metrics, to ensure that the cluster always has capacity to accommodate additional data.

You can also configure a "maximum size" for any shard when you add the shard using the `maxSize` parameter of the `addShard` (page 351) command. This will prevent the *balancer* from migrating chunks to the shard when the value of `mapped` exceeds the `maxSize` setting.

See also:

Change the Maximum Storage Size for a Given Shard (page 325) and *Monitoring Database Systems* (page 109).

Chunk Migration

MongoDB migrates chunks in a *sharded cluster* to distribute data evenly among shards. Migrations may be either:

- Manual. In these migrations you must specify the chunk that you want to migrate and the destination shard. Only migrate chunks manually after initiating sharding, to distribute data during bulk inserts, or if the cluster becomes uneven. See *Migrating Chunks* (page 323) for more details.
- Automatic. The balancer process handles most migrations when distribution of chunks between shards becomes uneven. See *Migration Thresholds* (page 308) for more details.

All chunk migrations use the following procedure:

1. The balancer process sends the `moveChunk` command to the source shard for the chunk. In this operation the balancer passes the name of the destination shard to the source shard.
2. The source initiates the move with an internal `moveChunk` command with the destination shard.
3. The destination shard begins requesting documents in the chunk, and begins receiving these chunks.
4. After receiving the final document in the chunk, the destination shard initiates a synchronization process to ensure that all changes to the documents in the chunk on the source shard during the migration process exist on the destination shard.

   When fully synchronized, the destination shard connects to the *config database* and updates the chunk location in the cluster metadata. After completing this operation, once there are no open cursors on the chunk, the source shard starts deleting its copy of documents from the migrated chunk.

If enabled, the `_secondaryThrottle` setting causes the balancer to wait for replication to secondaries. For more information, see *Require Replication before Chunk Migration (Secondary Throttle)* (page 326).

Detect Connections to `mongos` Instances

If your application must detect if the MongoDB instance its connected to is `mongos`, use the `isMaster` (page 289) command. When a client connects to a `mongos`, `isMaster` (page 289) 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.

Config Database

The `config` database contains information about your sharding configuration and stores the information in a set of collections used by sharding.

Important: Back up the `config` database before performing any maintenance on the config server.

To access the `config` database, issue the following command from the `mongo` shell:

```
use config
```

In general, you should *never* manipulate the content of the config database directly. The `config` database contains the following collections:

- `changelog`
- `chunks`

- collections
- databases
- lockpings
- locks
- mongos
- settings
- shards
- version

See http://docs.mongodb.org/manual/reference/config-database for full documentation of these collections and their role in sharded clusters.

Sharding GridFS Stores

When sharding a *GridFS* store, consider the following:

- 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.

- To shard the chunks collection by { files_id : 1 , n : 1 }, issue commands similar to the following:

  ```
  db.fs.chunks.ensureIndex( { files_id : 1 , n : 1 } )

  db.runCommand( { shardCollection : "test.fs.chunks" , key : { files_id : 1 , n : 1 } } )
  ```

  You may also want shard using just the file_id field, as in the following operation:

  ```
  db.runCommand( { shardCollection : "test.fs.chunks" , key : {  files_id : 1 } } )
  ```

  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 different value for _id in the files collection.

8.2 Sharded Cluster Tutorials and Procedures

The documents listed in this section address common sharded cluster operational practices in greater detail.

8.2.1 Getting Started With Sharded Clusters

Deploy a Sharded Cluster

The topics on this page present an ordered sequence of the tasks required to set up a *sharded cluster*. Before deploying a sharded cluster for the first time, consider the *Sharded Cluster Overview* (page 295) and *Sharded Cluster Architectures* (page 302) documents.

To set up a sharded cluster, complete the following sequence of tasks in the order defined below:

1. *Start the Config Server Database Instances* (page 312)
2. *Start the mongos Instances* (page 313)
3. *Add Shards to the Cluster* (page 313)
4. *Enable Sharding for a Database* (page 314)
5. *Enable Sharding for a Collection* (page 314)

Warning: Sharding and "localhost" Addresses
If you use either "localhost" or `127.0.0.1` as the hostname portion of any host identifier, for example as the `host` argument to `addShard` (page 351) 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.

Config server instances receive relatively little traffic and demand only a small portion of system resources. Therefore, you can run an instance on a system that runs other cluster components.

1. Create data directories for each of the three config server instances. By default, a config server stores its data files in the */data/configdb* directory. You can choose a different location. To create a data directory, issue a command similar to the following:

   ```
   mkdir /data/configdb
   ```

2. Start the three config server instances. Start each by issuing a command using the following syntax:

   ```
   mongod --configsvr --dbpath <path> --port <port>
   ```

   The default port for config servers is `27019`. You can specify a different port. The following example starts a config server using the default port and default data directory:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27019
   ```

   For additional command options, see `http://docs.mongodb.org/manual/reference/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. For operational flexibility, use DNS names for the config servers rather than explicit IP addresses. If you're not using resolvable hostname, you cannot change the config server names or IP addresses without a restarting *every* mongos and mongod instance.

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* 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.

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() (page 347) method, as shown in the examples below. Issue sh.addShard() (page 347) separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the addShard (page 351) database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see addShard (page 351).

   The following are examples of adding a shard with sh.addShard() (page 347):

- To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

  ```
  sh.addShard( "rs1/mongodb0.example.net:27017" )
  ```

  Changed in version 2.0.3.

  For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

  ```
  sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
  ```

- To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

  ```
  sh.addShard( "mongodb0.example.net:27017" )
  ```

Note: It might take some time for *chunks* to migrate to the new shard.

Enable Sharding for a Database

Before you can shard a collection, you must enable sharding for the collection's database. Enabling sharding for a database does not redistribute data but make it possible to shard the collections in that database.

Once you enable sharding for a database, MongoDB assigns a *primary shard* for that database where MongoDB stores all data before sharding begins.

1. From a `mongo` shell, connect to the `mongos` instance. Issue a command using the following syntax:

   ```
   mongo --host <hostname of machine running mongos> --port <port mongos listens on>
   ```

2. Issue the `sh.enableSharding()` (page 347) method, specifying the name of the database for which to enable sharding. Use the following syntax:

   ```
   sh.enableSharding("<database>")
   ```

Optionally, you can enable sharding for a database using the `enableSharding` (page 352) 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 *Shard Key Selection* (page 296).

2. Enable sharding for a collection by issuing the `sh.shardCollection()` (page 348) method in the `mongo` shell. The method uses the following syntax:

   ```
   sh.shardCollection("<database>.<collection>", shard-key-pattern)
   ```

   Replace the `<database>.<collection>` string with the full namespace of your database, which consists of the name of your database, a dot (e.g. `.`), and the full name of the collection. The `shard-key-pattern` represents your shard key, which you specify in the same form as you would an `index` key pattern.

Example

The following sequence of commands shards four collections:

```
sh.shardCollection("records.people", { "zipcode": 1, "name": 1 } )
sh.shardCollection("people.addresses", { "state": 1, "_id": 1 } )
sh.shardCollection("assets.chairs", { "type": 1, "_id": 1 } )
sh.shardCollection("events.alerts", { "hashed_id": 1 } )
```

In order, these operations shard:

1. The `people` collection in the `records` database using the shard key `{ "zipcode": 1, "name": 1 }`.

   This shard key distributes documents by the value of the `zipcode` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 305) by the values of the `name` field.

2. The `addresses` collection in the `people` database using the shard key `{ "state": 1, "_id": 1 }`.

   This shard key distributes documents by the value of the `state` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 305) by the values of the `_id` field.

3. The `chairs` collection in the `assets` database using the shard key `{ "type": 1, "_id": 1 }`.

   This shard key distributes documents by the value of the `type` field. If a number of documents have the same value for this field, then that *chunk* will be *splittable* (page 305) by the values of the `_id` field.

4. The `alerts` collection in the `events` database using the shard key `{ "hashed_id": 1 }`.

   This shard key distributes documents by the value of the `hashed_id` field. Presumably this is a computed value that holds the hash of some value in your documents and is able to evenly distribute documents throughout your cluster.

Add Shards to a Cluster

You add shards to a *sharded cluster* after you create the cluster or anytime that you need to add capacity to the cluster. If you have not created a sharded cluster, see *Deploy a Sharded Cluster* (page 312).

When adding a shard to a cluster, you should always ensure that the cluster has enough capacity to support the migration without affecting legitimate production traffic.

In production environments, all shards should be *replica sets*.

Add a Shard to a Cluster

You interact with a sharded cluster by connecting to a `mongos` instance.

1. From a `mongo` shell, connect to the `mongos` instance. 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()` (page 347) method, as shown in the examples below. Issue `sh.addShard()` (page 347) separately for each shard. If the shard is a replica set, specify the name of the replica set and specify a member of the set. In production deployments, all shards should be replica sets.

   Optional

   You can instead use the `addShard` (page 351) database command, which lets you specify a name and maximum size for the shard. If you do not specify these, MongoDB automatically assigns a name and maximum size. To use the database command, see `addShard` (page 351).

   The following are examples of adding a shard with `sh.addShard()` (page 347):

   - To add a shard for a replica set named `rs1` with a member running on port `27017` on `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017" )
     ```

     Changed in version 2.0.3.

     For MongoDB versions prior to 2.0.3, you must specify all members of the replica set. For example:

     ```
     sh.addShard( "rs1/mongodb0.example.net:27017,mongodb1.example.net:27017,mongodb2.example.net
     ```

   - To add a shard for a standalone `mongod` on port `27017` of `mongodb0.example.net`, issue the following command:

     ```
     sh.addShard( "mongodb0.example.net:27017" )
     ```

   Note: It might take some time for *chunks* to migrate to the new shard.

View Cluster Configuration

List Databases with Sharding Enabled

To list the databases that have sharding enabled, query the `databases` collection in the *config-database*. 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` (page 352) command, as follows:

```
use admin
db.runCommand( { listShards : 1 } )
```

View Cluster Details

To view cluster details, issue `db.printShardingStatus()` or `sh.status()` (page 349). Both methods return the same output.

Example

In the following example output from `sh.status()` (page 349)

- `sharding version` displays the version number of the shard metadata.
- `shards` displays a list of the `mongod` instances used as shards in the cluster.
- `databases` displays all databases in the cluster, including database that do not have sharding enabled.
- The `chunks` information for the `foo` database displays how many chunks are on each shard and displays the range of each chunk.

```
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
    {  "_id" : "shard0000",  "host" : "m0.example.net:30001" }
    {  "_id" : "shard0001",  "host" : "m3.example2.net:50000" }
  databases:
    {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
    {  "_id" : "animals",  "partitioned" : true,  "primary" : "shard0000" }
        foo.big chunks:
                shard0001    1
                shard0000    6
            { "a" : { $minKey : 1 } } -->> { "a" : "elephant" } on : shard0001 Timestamp(2000, 1) jur
            { "a" : "elephant" } -->> { "a" : "giraffe" } on : shard0000 Timestamp(1000, 1) jumbo
            { "a" : "giraffe" } -->> { "a" : "hippopotamus" } on : shard0000 Timestamp(2000, 2) jumbo
            { "a" : "hippopotamus" } -->> { "a" : "lion" } on : shard0000 Timestamp(2000, 3) jumbo
            { "a" : "lion" } -->> { "a" : "rhinoceros" } on : shard0000 Timestamp(1000, 3) jumbo
            { "a" : "rhinoceros" } -->> { "a" : "springbok" } on : shard0000 Timestamp(1000, 4)
            { "a" : "springbok" } -->> { "a" : { $maxKey : 1 } } on : shard0000 Timestamp(1000, 5)
        foo.large chunks:
                shard0001    1
                shard0000    5
            { "a" : { $minKey : 1 } } -->> { "a" : "hen" } on : shard0001 Timestamp(2000, 0)
            { "a" : "hen" } -->> { "a" : "horse" } on : shard0000 Timestamp(1000, 1) jumbo
            { "a" : "horse" } -->> { "a" : "owl" } on : shard0000 Timestamp(1000, 2) jumbo
            { "a" : "owl" } -->> { "a" : "rooster" } on : shard0000 Timestamp(1000, 3) jumbo
            { "a" : "rooster" } -->> { "a" : "sheep" } on : shard0000 Timestamp(1000, 4)
            { "a" : "sheep" } -->> { "a" : { $maxKey : 1 } } on : shard0000 Timestamp(1000, 5)
    {  "_id" : "test",  "partitioned" : false,  "primary" : "shard0000" }
```

8.2.2 Sharded Cluster Maintenance and Administration

Manage the Config Servers

Config servers (page 298) store all cluster metadata, most importantly, the mapping from *chunks* to *shards*. This section provides an overview of the basic procedures to migrate, replace, and maintain these servers.

This page includes the following:

- *Deploy Three Config Servers for Production Deployments* (page 318)
- *Migrate Config Servers with the Same Hostname* (page 318)
- *Migrate Config Servers with Different Hostnames* (page 319)
- *Replace a Config Server* (page 319)
- *Backup Cluster Metadata* (page 320)

Deploy Three Config Servers for Production Deployments

For redundancy, all production *sharded clusters* should deploy three config servers processes on three different machines.

Do not use only a single config server for production deployments. Only use a single config server deployments for testing. You should upgrade to three config servers immediately if you are shifting to production. The following process shows how to convert a test deployment with only one config server to production deployment with three config servers.

1. Shut down all existing MongoDB processes 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*, `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.

Migrate Config Servers with the Same Hostname

Use this process when you need to migrate a config server to a new system but the new system will be accessible using the same host name.

1. Shut down the config server that you're moving.

   This will render all config data for your cluster *read only* (page 298).

2. Change the DNS entry that points to the system that provided the old config server, so that the *same* hostname points to the new system.

   How you do this depends on how you organize your DNS and hostname resolution services.

3. Move the entire `dbpath` file system tree from the old config server to the new config server. This command, issued on the old config server system, may resemble the following:

   ```
   rsync -az /data/configdb mongo-config0.example.net:/data/configdb
   ```

4. Start the config instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

When you start the third config server, your cluster will become writable and it will be able to create new splits and migrate chunks as needed.

Migrate Config Servers with Different Hostnames

Use this process when you need to migrate a *config-database* to a new server and it *will not* be accessible via the same hostname. If possible, avoid changing the hostname so that you can use the *previous procedure* (page 318).

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 328) for more information.
2. Shut down the *config server* (page 298) you're moving.

   This will render all config data for your cluster "read only:"

   ```
   rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
   ```

3. Start the config instance on the new system. The default invocation is:

   ```
   mongod --configsvr
   ```

4. Shut down all existing MongoDB processes. This includes:
   - all `mongod` instances or *replica sets* that provide your shards.
   - the `mongod` instances that provide your existing *config databases*.
   - all `mongos` instances in your cluster.
5. Restart all `mongod` processes that provide the shard servers.
6. Update the *`--configdb`* parameter (or `configdb`) for all `mongos` instances and restart all `mongos` instances.
7. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 328) section for more information on managing the balancer process.

Replace a Config Server

Use this procedure only if you need to replace one of your config servers after it becomes inoperable (e.g. hardware failure.) This process assumes that the hostname of the instance will not change. If you must change the hostname of the instance, use the process for *migrating a config server to a different hostname* (page 319).

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 328) for more information.
2. Provision a new system, with the same hostname as the previous host.

   You will have to ensure that the new system has the same IP address and hostname as the system it's replacing *or* you will need to modify the DNS records and wait for them to propagate.

3. Shut down *one* (and only one) of the existing config servers. Copy all this host's `dbpath` file system tree from the current system to the system that will provide the new config server. This command, issued on the system with the data files, may resemble the following:

```
rsync -az /data/configdb mongodb.config2.example.net:/data/configdb
```

4. Restart the config server process that you used in the previous step to copy the data files to the new config server instance.
5. Start the new config server instance. The default invocation is:

   ```
   mongod --configsvr
   ```

6. Re-enable the balancer to allow the cluster to resume normal balancing operations. See the *Disable the Balancer* (page 328) section for more information on managing the balancer process.

Note: In the course of this procedure *never* remove a config server from the `configdb` parameter on any of the `mongos` instances. If you need to change the name of a config server, always make sure that all `mongos` instances have three config servers specified in the `configdb` setting at all times.

Backup Cluster Metadata

The cluster will remain operational [7] without one of the config database's `mongod` instances, creating a backup of the cluster metadata from the config database is straight forward:

1. Disable the cluster balancer process temporarily. See *Disable the Balancer* (page 328) 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 328) section for more information on managing the balancer process.

See also:

Backup Strategies for MongoDB Systems (page 120).

Manage Chunks in a Sharded Cluster

This page describes various operations on *chunks* in *sharded clusters*. MongoDB automates most chunk management operations. However, these chunk management operations are accessible to administrators for use in some situations, typically surrounding initial setup, deployment, and data ingestion.

Split Chunks

Normally, MongoDB splits a *chunk* following inserts when a chunk exceeds the *chunk size* (page 309). The *balancer* may migrate recently split chunks to a new shard immediately if `mongos` predicts future insertions will benefit from the move.

MongoDB treats all chunks the same, whether split manually or automatically by the system.

Warning: You cannot merge or combine chunks once you have split them.

[7] 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. The *Config Servers* (page 298) section of the documentation provides more information on this topic.

You may want to split chunks manually if:

- you have a large amount of data in your cluster and very few *chunks*, as is the case after deploying a cluster using existing data.
- you expect to add a large amount of data that would initially reside in a single chunk or shard.

Example

You plan to insert a large amount of data with *shard key* values between `300` and `400`, *but* all values of your shard keys are between `250` and `500` are in a single chunk.

Warning: Be careful when splitting data in a sharded collection to create new chunks. When you shard a collection that has existing data, MongoDB automatically creates chunks to evenly distribute the collection. To split data effectively in a sharded cluster you must consider the number of documents in a chunk and the average document size to create a uniform chunk size. When chunks have irregular sizes, shards may have an equal number of chunks but have very different data sizes. Avoid creating splits that lead to a collection with differently sized chunks.

Use `sh.status()` (page 349) to determine the current chunks ranges across the cluster.

To split chunks manually, use the `split` command with operators: `middle` and `find`. The equivalent shell helpers are `sh.splitAt()` (page 348) or `sh.splitFind()` (page 348).

Example

The following command will split the chunk that contains the value of `63109` for the `zipcode` field in the `people` collection of the `records` database:

```
sh.splitFind( "records.people", { "zipcode": 63109 } )
```

`sh.splitFind()` (page 348) will split the chunk that contains the *first* document returned that matches this query into two equally sized chunks. You must specify the full namespace (i.e. "`<database>.<collection>`") of the sharded collection to `sh.splitFind()` (page 348). The query in `sh.splitFind()` (page 348) need not contain the shard key, though it almost always makes sense to query for the shard key in this case, and including the shard key will expedite the operation.

Use `sh.splitAt()` (page 348) to split a chunk in two using the queried document as the partition point:

```
sh.splitAt( "records.people", { "zipcode": 63109 } )
```

However, the location of the document that this query finds with respect to the other documents in the chunk does not affect how the chunk splits.

Create Chunks (Pre-Splitting)

Pre-splitting the chunk ranges in an empty sharded collection, allows clients to insert data into an already-partitioned collection. In most situations a *sharded cluster* will create and distribute chunks automatically without user intervention. However, in a limited number of use profiles, MongoDB cannot create enough chunks or distribute data fast enough to support required throughput. For example, if:

- you must partition an existing data collection that resides on a single shard.
- you must ingest a large volume of data into a cluster that isn't balanced, or where the ingestion of data will lead to an imbalance of data.

This can arise in an initial data loading, or in a case where you must insert a large volume of data into a single chunk, as is the case when you must insert at the beginning or end of the chunk range, as is the case for monotonically increasing or decreasing shard keys.

Preemptively splitting chunks increases cluster throughput for these operations, by reducing the overhead of migrating chunks that hold data during the write operation. MongoDB only creates splits after an insert operation and can migrate only a single chunk at a time. Chunk migrations are resource intensive and further complicated by large write volume to the migrating chunk.

Warning: You can only pre-split an empty collection. When you enable sharding for a collection that contains data MongoDB automatically creates splits. Subsequent attempts to create splits manually, can lead to unpredictable chunk ranges and sizes as well as inefficient or ineffective balancing behavior.

To create and migrate chunks manually, use the following procedure:

1. Split empty chunks in your collection by manually performing `split` command on chunks.

   Example

   To create chunks for documents in the `myapp.users` collection, using the `email` field as the *shard key*, use the following operation in the `mongo` shell:

   ```
   for ( var x=97; x<97+26; x++ ){
     for( var y=97; y<97+26; y+=6 ) {
       var prefix = String.fromCharCode(x) + String.fromCharCode(y);
       db.runCommand( { split : "myapp.users" , middle : { email : prefix } } );
     }
   }
   ```

   This assumes a collection size of 100 million documents.

2. Migrate chunks manually using the `moveChunk` command:

   Example

   To migrate all of the manually created user profiles evenly, putting each prefix chunk on the next shard from the other, run the following commands in the mongo shell:

   ```
   var shServer = [ "sh0.example.net", "sh1.example.net", "sh2.example.net", "sh3.example.net",
   for ( var x=97; x<97+26; x++ ){
     for( var y=97; y<97+26; y+=6 ) {
       var prefix = String.fromCharCode(x) + String.fromCharCode(y);
       db.adminCommand({moveChunk : "myapp.users", find : {email : prefix}, to : shServer[(y-97
     }
   }
   ```

   You can also let the balancer automatically distribute the new chunks. For an introduction to balancing, see *Shard Balancing* (page 296). For lower level information on balancing, see *Cluster Balancer* (page 308).

Modify Chunk Size

When you initialize a sharded cluster, [8] the default chunk size is 64 megabytes. This default chunk size works well for most deployments; however, if you notice that automatic migrations are incurring a level of I/O that your hardware cannot handle, you may want to reduce the chunk size. For the automatic splits and migrations, a small chunk size leads to more rapid and frequent migrations.

[8] The first `mongos` that connects to a set of *config servers* initializes the sharded cluster.

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*:

   ```
   use config
   ```

3. Issue the following `save()` operation:

   ```
   db.settings.save( { _id:"chunksize", value: <size> } )
   ```

   Where the value of `<size>` reflects the new chunk size in megabytes. Here, you're essentially writing a document whose values store the global chunk size configuration value.

Note: The `chunkSize` and `--chunkSize` options, passed at runtime to the `mongos` **do not** affect the chunk size after you have initialized the cluster. [1]

To eliminate confusion you should *always* set chunk size using the above procedure and never use the runtime options.

Modifying the chunk size has several limitations:

- Automatic splitting only occurs when inserting *documents* or updating existing documents.
- If you lower the chunk size it may take time for all chunks to split to the new size.
- Splits cannot be "undone."

If you increase the chunk size, existing chunks must grow through insertion or updates until they reach the new size.

Migrate Chunks

In most circumstances, you should let the automatic balancer migrate *chunks* between *shards*. However, you may want to migrate chunks manually in a few cases:

- If you create chunks by *pre-splitting* the data in your collection, you will have to migrate chunks manually to distribute chunks evenly across the shards. Use pre-splitting in limited situations, to support bulk data ingestion.
- If the balancer in an active cluster cannot distribute chunks within the balancing window, then you will have to migrate chunks manually.

For more information on how chunks move between shards, see *Cluster Balancer* (page 308), in particular the section *Chunk Migration* (page 309).

To migrate chunks, use the `moveChunk` command.

Note: To return a list of shards, use the `listShards` (page 352) command.

Specify shard names using the `addShard` (page 351) command using the `name` argument. If you do not specify a name in the `addShard` (page 351) command, MongoDB will assign a name automatically.

The following example assumes that the field `username` is the *shard key* for a collection named `users` in the `myapp` database, and that the value `smith` exists within the *chunk* you want to migrate.

To move this chunk, you would issue the following command from a `mongo` shell connected to any `mongos` instance.

```
db.adminCommand( { moveChunk : "myapp.users",
                   find : {username : "smith"},
                   to : "mongodb-shard3.example.net" } )
```

This command moves the chunk that includes the shard key value "smith" to the *shard* named `mongodb-shard3.example.net`. The command will block until the migration is complete.

See *Create Chunks (Pre-Splitting)* (page 321) for an introduction to pre-splitting.

New in version 2.2: `moveChunk` command has the: `_secondaryThrottle` parameter. When set to `true`, MongoDB ensures that changes to shards as part of chunk migrations replicate to *secondaries* throughout the migration operation. For more information, see *Require Replication before Chunk Migration (Secondary Throttle)* (page 326).

Warning: The `moveChunk` command may produce the following error message:

```
The collection's metadata lock is already taken.
```

These errors occur when clients have too many open *cursors* that access the chunk you are migrating. You can either wait until the cursors complete their operation or close the cursors manually.

Strategies for Bulk Inserts in Sharded Clusters

Large bulk insert operations, including initial data ingestion or routine data import, can have a significant impact on a *sharded cluster*. For bulk insert operations, consider the following strategies:

- If the collection does not have data, then there is only one *chunk*, which must reside on a single shard. MongoDB must receive data, create splits, and distribute chunks to the available shards. To avoid this performance cost, you can pre-split the collection, as described in *Create Chunks (Pre-Splitting)* (page 321).
- You can parallelize import processes by sending insert operations to more than one `mongos` instance. If the collection is empty, pre-split first, as described in *Create Chunks (Pre-Splitting)* (page 321).
- If your shard key increases monotonically during an insert then all the inserts will go to the last chunk in the collection, which will always end up on a single shard. Therefore, the insert capacity of the cluster will never exceed the insert capacity of a single shard.

  If your insert volume is never larger than what a single shard can process, then there is no problem; however, if the insert volume exceeds that range, and you cannot avoid a monotonically increasing shard key, then consider the following modifications to your application:

  - Reverse all the bits of the shard key to preserve the information while avoiding the correlation of insertion order and increasing sequence of values.
  - Swap the first and last 16-bit words to "shuffle" the inserts.

  Example

  The following example, in C++, swaps the leading and trailing 16-bit word of *BSON ObjectIds* generated so that they are no longer monotonically increasing.

```
using namespace mongo;
OID make_an_id() {
  OID x = OID::gen();
  const unsigned char *p = x.getData();
  swap( (unsigned short&) p[0], (unsigned short&) p[10] );
  return x;
}

void foo() {
  // create an object
  BSONObj o = BSON( "_id" << make_an_id() << "x" << 3 << "name" << "jane" );
  // now we might insert o into a sharded collection...
}
```

For information on choosing a shard key, see *Shard Key Selection* (page 296) and see *Shard Key Internals* (page 304) (in particular, *Operations and Reliability* (page 306) and *Choosing a Shard Key* (page 307)).

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.

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 *Shard Balancing* (page 296) and *Cluster Balancer* (page 308).

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 327)
- *Remove a Balancing Window Schedule* (page 328).

The `mongos` instances user their own local timezones to when respecting balancer window.

Configure Default Chunk Size

The default chunk size for a sharded cluster is 64 megabytes. In most situations, the default size is appropriate for splitting and migrating chunks. For information on how chunk size affects deployments, see details, see *Chunk Size* (page 309).

Changing the default chunk size affects chunks that are processes during migrations and auto-splits but does not retroactively affect all chunks.

To configure default chunk size, see *Modify Chunk Size* (page 322).

Change the Maximum Storage Size for a Given Shard

The `maxSize` field in the `shards` collection in the *config database* sets the maximum size for a shard, allowing you to control whether the balancer will migrate chunks to a shard. If `dataSize` is above a shard's `maxSize`, the balancer will not move chunks to the shard. Also, the balancer will not move chunks off an overloaded shard. This must happen manually. The `maxSize` value only affects the balancer's selection of destination shards.

By default, `maxSize` is not specified, allowing shards to consume the total amount of available space on their machines if necessary.

You can set `maxSize` both when adding a shard and once a shard is running.

To set `maxSize` when adding a shard, set the `addShard` (page 351) command's `maxSize` parameter to the maximum size in megabytes. For example, the following command run in the `mongo` shell adds a shard with a maximum size of 125 megabytes:

```
db.runCommand( { addshard : "example.net:34008", maxSize : 125 } )
```

To set `maxSize` on an existing shard, insert or update the `maxSize` field in the `shards` collection in the *config database*. Set the `maxSize` in megabytes.

Example

Assume you have the following shard without a `maxSize` field:

```
{ "_id" : "shard0000", "host" : "example.net:34001" }
```

Run the following sequence of commands in the `mongo` shell to insert a `maxSize` of 125 megabytes:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 125 } } )
```

To later increase the `maxSize` setting to 250 megabytes, run the following:

```
use config
db.shards.update( { _id : "shard0000" }, { $set : { maxSize : 250 } } )
```

Require Replication before Chunk Migration (Secondary Throttle)

New in version 2.2.1.

By default, the write operations required to migrate chunks between shards do not need to replicate to secondaries in order to succeed. However, you can configure the balancer to require migration related write operations to replicate to secondaries. This throttles or slows the migration process and in doing so reduces the potential impact of migrations on a sharded cluster.

You can throttle migrations by enabling the balancer's `_secondaryThrottle` parameter. When enabled, secondary throttle requires a `{ w : 2 }` write concern on delete and insertion operations, so that every operation propagates to at least one secondary before the balancer issues the next operation.

You enable `_secondaryThrottle` directly in the `settings` collection in the *config database* by running the following commands from the `mongo` shell:

```
use config
db.settings.update( { "_id" : "balancer" } , { $set : { "_secondaryThrottle" : true } , { upsert : tr
```

You also can enable secondary throttle when issuing the `moveChunk` command by setting `_secondaryThrottle` to `true`. For more information, see `moveChunk`.

Manage Sharded Cluster Balancer

This page describes provides common administrative procedures related to balancing. For an introduction to balancing, see *Shard Balancing* (page 296). For lower level information on balancing, see *Cluster Balancer* (page 308).

See also:

Configure Behavior of Balancer Process in Sharded Clusters (page 325)

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*:

   ```
   use config
   ```

3. Use the following query to return the balancer lock:

   ```
   db.locks.find( { _id : "balancer" } ).pretty()
   ```

When this command returns, you will see output like the following:

```
{   "_id" : "balancer",
"process" : "mongos0.example.net:1292810611:1804289383",
  "state" : 2,
     "ts" : ObjectId("4d0f872630c42d1978be8a2e"),
   "when" : "Mon Dec 20 2010 11:41:10 GMT-0500 (EST)",
    "who" : "mongos0.example.net:1292810611:1804289383:Balancer:846930886",
    "why" : "doing balance round" }
```

This output confirms that:

- The balancer originates from the `mongos` running on the system with the hostname `mongos0.example.net`.
- The value in the `state` field indicates that a `mongos` has the lock. For version 2.0 and later, the value of an active lock is `2`; for earlier versions the value is `1`.

Optional

You can also use the following shell helper, which returns a boolean to report if the balancer is active:

```
sh.getBalancerState()
```

Schedule the Balancing Window

In some situations, particularly when your data set grows slowly and a migration can impact performance, it's useful to be able to ensure that the balancer is active only at certain times. Use the following procedure to specify a window during which the *balancer* will be able to migrate chunks:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *config-database*:

   ```
   use config
   ```

3. Use an operation modeled on the following example `update()` operation to modify the balancer's window:

   ```
   db.settings.update({ _id : "balancer" }, { $set : { activeWindow : { start : "<start-time>", sto
   ```

   Replace `<start-time>` and `<end-time>` with time values using two digit hour and minute values (e.g `HH:MM`) that describe the beginning and end boundaries of the balancing window. These times will be evaluated relative to the time zone of each individual `mongos` 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:
   ```

Note: The balancer window must be sufficient to *complete* the migration of all data inserted during the day.

As data insert rates can change based on activity and usage patterns, it is important to ensure that the balancing window you select will be sufficient to support the needs of your deployment.

Remove a Balancing Window Schedule

If you have *set the balancing window* (page 327) and wish to remove the schedule so that the balancer is always running, issue the following sequence of operations:

```
use config
db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

Disable the Balancer

By default the balancer may run at any time and only moves chunks as needed. To disable the balancer for a short period of time and prevent all migration, use the following procedure:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue *one* of the following operations to disable the balancer:

   ```
   sh.stopBalancer()
   ```

3. Later, issue *one* the following operations to enable the balancer:

   ```
   sh.startBalancer()
   ```

Note: If a migration is in progress, the system will complete the in-progress migration. After disabling, you can use the following operation in the `mongo` shell to determine if there are no migrations in progress:

```
use config
while( db.locks.findOne({_id: "balancer"}).state ) {
       print("waiting..."); sleep(1000);
}
```

The above process and the `sh.setBalancerState()` (page 349), `sh.startBalancer()`, and `sh.stopBalancer()` helpers provide wrappers on the following process, which may be useful if you need to run this operation from a driver that does not have helper functions:

1. Connect to any `mongos` in the cluster using the `mongo` shell.
2. Issue the following command to switch to the *config-database*:

   ```
   use config
   ```

3. Issue the following update to disable the balancer:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: true } } , true );
   ```

4. To enable the balancer again, alter the value of "stopped" as follows:

   ```
   db.settings.update( { _id: "balancer" }, { $set : { stopped: false } } , true );
   ```

Disable Balancing During Backups

If MongoDB migrates a *chunk* during a *backup* (page 120), 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 327) 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 328) for the duration of the backup procedure.

Confirm that the balancer is not active using the `sh.getBalancerState()` method before starting a backup operation. When the backup procedure is complete you can reactivate the balancer process.

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 follow the ordered sequence of tasks on this page:

1. *Ensure the Balancer Process is Active* (page 329)
2. *Determine the Name of the Shard to Remove* (page 329)
3. *Remove Chunks from the Shard* (page 330)
4. *Check the Status of the Migration* (page 330)
5. *Move Unsharded Data* (page 330)
6. *Finalize the Migration* (page 331)

Ensure the Balancer Process is Active

To successfully migrate data from a shard, the *balancer* process **must** be active. Check the balancer state using the `sh.getBalancerState()` helper in the `mongo` shell. For more information, see the section on *balancer operations* (page 328).

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` (page 352) command, as in the following:

  ```
  db.adminCommand( { listShards: 1 } )
  ```

- Run either the `sh.status()` (page 349) or the `db.printShardingStatus()` method.

The `shards._id` field lists the name of each shard.

Remove Chunks from the Shard

Run the `removeShard` (page 354) command. This begins "draining" chunks from the shard you are removing to other shards in the cluster. For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

This operation returns immediately, with the following response:

```
{ msg : "draining started successfully" , state: "started" , shard :"mongodb0" , ok : 1 }
```

Depending on your network capacity and the amount of data, this operation can take from a few minutes to several days to complete.

Check the Status of the Migration

To check the progress of the migration at any stage in the process, run `removeShard` (page 354). For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

The command returns output similar to the following:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: NumberLong(42), dbs : NumberLong
```

In the output, the `remaining` document displays the remaining number of chunks that MongoDB must migrate to other shards and the number of MongoDB databases that have "primary" status on this shard.

Continue checking the status of the *removeShard* command until the number of chunks remaining is `0`. Then proceed to the next step.

Move Unsharded Data

If the shard is the *primary shard* for one or more databases in the cluster, then the shard will have unsharded data. If the shard is not the primary shard for any databases, skip to the next task, *Finalize the Migration* (page 331).

In a cluster, a database with unsharded collections stores those collections only on a single shard. That shard becomes the primary shard for that database. (Different databases in a cluster can have different primary shards.)

Warning: Do not perform this procedure until you have finished draining the shard.

1. To determine if the shard you are removing is the primary shard for any of the cluster's databases, issue one of the following methods:

   - `sh.status()` (page 349)
   - `db.printShardingStatus()`

   In the resulting document, the `databases` field lists each database and its primary shard. For example, the following `database` field shows that the `products` database uses `mongodb0` as the primary shard:

   ```
   {  "_id" : "products",  "partitioned" : true,  "primary" : "mongodb0" }
   ```

2. To move a database to another shard, use the `movePrimary` command. For example, to migrate all remaining unsharded data from `mongodb0` to `mongodb1`, issue the following command:

   ```
   db.runCommand( { movePrimary: "products", to: "mongodb1" })
   ```

This command does not return until MongoDB completes moving all data, which may take a long time. The response from this command will resemble the following:

```
{ "primary" : "mongodb1", "ok" : 1 }
```

Finalize the Migration

To clean up all metadata information and finalize the removal, run `removeShard` (page 354) again. For example, for a shard named `mongodb0`, run:

```
db.runCommand( { removeShard: "mongodb0" } )
```

A success message appears at completion:

```
{ msg: "remove shard completed successfully" , stage: "completed", host: "mongodb0", ok : 1 }
```

Once the value of the `stage` field is "completed", you may safely stop the processes comprising the `mongodb0` shard.

8.2.3 Backup and Restore Sharded Clusters

Backup a Small Sharded Cluster with `mongodump`

Overview

If your *sharded cluster* holds a small data set, you can connect to a `mongos` using `mongodump`. You can create backups of your MongoDB cluster, if your backup infrastructure can capture the entire backup in a reasonable amount of time and if you have a storage system that can hold the complete MongoDB data set.

Read *Sharded Cluster Backup Considerations* (page 121) for a high-level overview of important considerations as well as a list of alternate backup tutorials.

Important: By default `mongodump` issue its queries to the non-primary nodes.

Procedure

Capture Data

Note: If you use `mongodump` without specifying a database or collection, `mongodump` will capture collection data *and* the cluster meta-data from the *config servers* (page 298).

You cannot use the `--oplog` option for `mongodump` when capturing data from `mongos`. This option is only available when running directly against a *replica set* member.

You can perform a backup of a *sharded cluster* by connecting `mongodump` to a `mongos`. Use the following operation at your system's prompt:

```
mongodump --host mongos3.example.net --port 27017
```

`mongodump` will write *BSON* files that hold a copy of data stored in the *sharded cluster* accessible via the `mongos` listening on port `27017` of the `mongos3.example.net` host.

Restore Data Backups created with `mongodump` do not reflect the chunks or the distribution of data in the sharded collection or collections. Like all `mongodump` output, these backups contain separate directories for each database and *BSON* files for each collection in that database.

You can restore `mongodump` output to any MongoDB instance, including a standalone, a *replica set*, or a new *sharded cluster*. When restoring data to sharded cluster, you must deploy and configure sharding before restoring data from the backup. See *Deploy a Sharded Cluster* (page 312) for more information.

Create Backup of a Sharded Cluster with Filesystem Snapshots

Overview

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses file system snapshots to capture a copy of the `mongod` instance. An alternate procedure that uses `mongodump` to create binary database dumps when file-system snapshots are not available. See *Create Backup of a Sharded Cluster with Database Dumps* (page 333) for the alternate procedure.

See *Sharded Cluster Backup Considerations* (page 121) for a full higher level overview backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Procedure

In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using a file-system snapshot tool. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment in time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

1. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell, and see the *Disable the Balancer* (page 328) procedure.

   > **Warning:** It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* may migrate while recording backups.

2. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` instances in as short of an interval as possible.

   To lock or freeze a sharded cluster, you must:

   - use the `db.fsyncLock()` method in the `mongo` shell connected to a single secondary member of the replica set that provides shard `mongod` instance.
   - Shutdown one of the *config servers* (page 298), to prevent all metadata changes during the backup process.

3. Use `mongodump` to backup one of the *config servers* (page 298). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

   Issue this command against one of the config `mongod` instances or via the `mongos`:

```
mongodump --db config
```

4. Back up the replica set members of the shards that you locked. You may back up the shards in parallel. For each shard, create a snapshot. Use the procedures in http://docs.mongodb.org/manual/tutorial/backup-databases-with-filesystem-snapshots.
5. Unlock all locked replica set members of each shard using the `db.fsyncUnlock()` method in the `mongo` shell.
6. Re-enable the balancer with the `sh.setBalancerState()` (page 349) method.

   Use the following command sequence when connected to the `mongos` with the `mongo` shell:

   ```
   use config
   sh.setBalancerState(true)
   ```

Create Backup of a Sharded Cluster with Database Dumps

Overview

This document describes a procedure for taking a backup of all components of a sharded cluster. This procedure uses `mongodump` 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 *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 332).

See *Sharded Cluster Backup Considerations* (page 121) for a full higher level overview of backing up a sharded cluster as well as links to other tutorials that provide alternate procedures.

Important: To capture a point-in-time backup from a sharded cluster you **must** stop *all* writes to the cluster. On a running production system, you can only capture an *approximation* of point-in-time snapshot.

Procedure

In this procedure, you will stop the cluster balancer and take a backup up of the *config database*, and then take backups of each shard in the cluster using `mongodump` to capture the backup data. If you need an exact moment-in-time snapshot of the system, you will need to stop all application writes before taking the filesystem snapshots; otherwise the snapshot will only approximate a moment of time.

For approximate point-in-time snapshots, you can improve the quality of the backup while minimizing impact on the cluster by taking the backup from a secondary member of the replica set that provides each shard.

1. Disable the *balancer* process that equalizes the distribution of data among the *shards*. To disable the balancer, use the `sh.stopBalancer()` method in the `mongo` shell, and see the *Disable the Balancer* (page 328) procedure.

   > **Warning:** It is essential that you stop the balancer before creating backups. If the balancer remains active, your resulting backups could have duplicate data or miss some data, as *chunks* migrate while recording backups.

2. Lock one member of each replica set in each shard so that your backups reflect the state of your database at the nearest possible approximation of a single moment in time. Lock these `mongod` instances in as short of an interval as possible.

   To lock or freeze a sharded cluster, you must:

- Shutdown one member of each replica set.

  Ensure that the *oplog* has sufficient capacity to allow these secondaries to catch up to the state of the primaries after finishing the backup procedure. See *Oplog* (page 220) for more information.

- Shutdown one of the *config servers* (page 298), to prevent all metadata changes during the backup process.

3. Use `mongodump` to backup one of the *config servers* (page 298). This backs up the cluster's metadata. You only need to back up one config server, as they all hold the same data.

   Issue this command against one of the config `mongod` instances or via the `mongos`:

   ```
   mongodump --journal --db config
   ```

4. Back up the replica set members of the shards that shut down using `mongodump` and specifying the `--dbpath` option. You may back up the shards in parallel. Consider the following invocation:

   ```
   mongodump --journal --dbpath /data/db/ --out /data/backup/
   ```

   You must run this command on the system where the `mongod` ran. This operation will use journaling and create a dump of the entire `mongod` instance with data files stored in `/data/db/`. `mongodump` will write the output of this dump to the `/data/backup/` directory.

5. Restart all stopped replica set members of each shard as normal and allow them to catch up with the state of the primary.

6. Re-enable the balancer with the `sh.setBalancerState()` (page 349) method.

   Use the following command sequence when connected to the `mongos` with the `mongo` shell:

   ```
   use config
   sh.setBalancerState(true)
   ```

Restore a Single Shard

Overview

Restoring a single shard from backup with other unaffected shards requires a number of special considerations and practices. This document outlines the additional tasks you must perform when restoring a single shard.

Consider the following resources on backups in general as well as backup and restoration of sharded clusters specifically:

- *Sharded Cluster Backup Considerations* (page 121)
- *Restore Sharded Clusters* (page 335)
- *Backup Strategies for MongoDB Systems* (page 120)

Procedure

Always restore *sharded clusters* as a whole. When you restore a single shard, keep in mind that the *balancer* process might have moved *chunks* to or from this shard since the last backup. If that's the case, you must manually move those chunks, as described in this procedure.

1. Restore the shard as you would any other `mongod` instance. See *Backup Strategies for MongoDB Systems* (page 120) for overviews of these procedures.

2. For all chunks that migrate away from this shard, you do not need to do anything at this time. You do not need to delete these documents from the shard because the chunks are automatically filtered out from queries by `mongos`. You can remove these documents from the shard, if you like, at your leisure.
3. For chunks that migrate to this shard after the most recent backup, you must manually recover the chunks using backups of other shards, or some other source. To determine what chunks have moved, view the `changelog` collection in the *config-database*.

Restore Sharded Clusters

Overview

The procedure outlined in this document addresses how to restore an entire sharded cluster. For information on related backup procedures consider the following tutorials which describe backup procedures in greater detail:

- *Create Backup of a Sharded Cluster with Filesystem Snapshots* (page 332)
- *Create Backup of a Sharded Cluster with Database Dumps* (page 333)

The exact procedure used to restore a database depends on the method used to capture the backup. See the *Backup Strategies for MongoDB Systems* (page 120) document for an overview of backups with MongoDB, as well as *Sharded Cluster Backup Considerations* (page 121) which provides an overview of the high level concepts important for backing up sharded clusters.

Procedure

1. Stop all `mongod` and `mongos` processes.
2. If shard hostnames have changed, you must manually update the `shards` collection in the *config-database* to use the new hostnames. Do the following:
   (a) Start the three *config servers* (page 298) by issuing commands similar to the following, using values appropriate to your configuration:

   ```
   mongod --configsvr --dbpath /data/configdb --port 27019
   ```

   (b) Restore the *config-database* on each config server.
   (c) Start one `mongos` instance.
   (d) Update the *config-database* collection named `shards` to reflect the new hostnames.
3. Restore the following:
   - Data files for each server in each *shard*. Because replica sets provide each production shard, restore all the members of the replica set or use the other standard approaches for restoring a replica set from backup. See the *backup-restore-snapshot* and *backup-restore-dump* sections for details on these procedures.
   - Data files for each *config server* (page 298), if you have not already done so in the previous step.
4. Restart all the `mongos` instances.
5. Restart all the `mongod` instances.
6. Connect to a `mongos` instance from a `mongo` shell and use the `db.printShardingStatus()` method to ensure that the cluster is operational, as follows:

   ```
   db.printShardingStatus()
   show collections
   ```

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 328) you can manually stop the balancer process temporarily. As an alternative you can use this procedure to define a balancing window so that the balancer is always disabled during your automated backup operation.

Procedure

If you have an automated backup schedule, you can disable all balancing operations for a period of time. For instance, consider the following command:

```
use config
db.settings.update( { _id : "balancer" }, { $set : { activeWindow : { start : "6:00", stop : "23:00"
```

This operation configures the balancer to run between 6:00am and 11:00pm, server time. Schedule your backup operation to run *and complete* outside of this time. Ensure that the backup can complete outside the window when the balancer is running *and* that the balancer can effectively balance the collection among the shards in the window allotted to each.

8.2.4 Application Development Patterns for Sharded Clusters

The following documents describe processes that application developers may find useful when developing applications that use data stored in a MongoDB sharded cluster. For some cases you will also want to consider the documentation of `/data-center-awareness`.

Tag Aware Sharding

For sharded clusters, MongoDB makes it possible to associate specific ranges of a *shard key* with a specific *shard* or subset of shards. This association dictates the policy of the cluster balancer process as it balances the *chunks* around the cluster. This capability enables the following deployment patterns:

- isolating a specific subset of data on specific set of shards.
- controlling the balancing policy so that in a geographically distributed cluster the most relevant portions of the data set reside on the shards with greatest proximity to the application servers.

This document describes the behavior, operation, and use of tag aware sharding in MongoDB deployments.

Note: Shard key range tags are entirely distinct from *replica set member tags* (page 246).

Behavior and Operations

Tags in a sharded cluster are pieces of metadata that dictate the policy and behavior of the cluster *balancer*. Using tags, you may associate individual shards in a cluster with one or more tags. Then, you can assign this tag string to

a range of *shard key* values for a sharded collection. When migrating a chunk, the balancer will select a destination shard based on the configured tag ranges.

The balancer migrates chunks in tagged ranges to shards with those tags, if tagged shards are not balanced. [9]

Note: Because a single chunk may span different tagged shard key ranges, the balancer may migrate chunks to tagged shards that contain values that exceed the upper bound of the selected tag range.

Example

Given a sharded collection with two configured tag ranges, such that:

- *Shard key* values between `100` and `200` have tags to direct corresponding chunks to shards tagged `NYC`.
- Shard Key values between `200` and `300` have tags to direct corresponding chunks to shards tagged `SFO`.

In this cluster, the balancer will migrate a chunk with shard key values ranging between `150` and `220` to a shard tagged `NYC`, since `150` is closer to `200` than `300`.

After configuring tags on shards and ranges of the shard key, the cluster may take some time to reach the proper distribution of data, depending on the division of chunks (i.e. splits) and the current distribution of data in the cluster. Once configured, the balancer will respect tag ranges during future *balancing rounds* (page 308).

Administer Shard Tags

Associate tags with a particular shard using the `sh.addShardTag()` (page 350) method when connected to a `mongos` instance. A single shard may have multiple tags, and multiple shards may also have the same tag.

Example

The following example adds the tag `NYC` to two shards, and the tags `SFO` and `NRT` to a third shard:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "NYC")
sh.addShardTag("shard0002", "SFO")
sh.addShardTag("shard0002", "NRT")
```

You may remove tags from a particular shard using the `sh.removeShardTag()` (page 351) method when connected to a `mongos` instance, as in the following example, which removes the `NRT` tag from a shard:

```
sh.removeShardTag("shard0002", "NRT")
```

Tag a Shard Key Range To assign a tag to a range of shard keys use the `sh.addTagRange()` (page 350) 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

[9] To migrate chunks in a tagged environment, the balancer selects a target shard with a tag range that has an *upper* bound that is *greater than* the migrating chunk's *lower* bound. If a shard with a matching tagged range exists, the balancer will migrate the chunk to that shard.

```
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 rages 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` collection of the `config` database.

Each document in the `tags` holds the *namespace* of the sharded collection and a minimum shard key value.

Example

The following example removes the `NYC` tag assignment for the range of zip codes within Manhattan:

```
use config
db.tags.remove({ _id: { ns: "records.users", min: { zipcode: "10001" }}, tag: "NYC" })
```

View Existing Shard Tags The output from `sh.status()` (page 349) lists tags associated with a shard, if any, for each shard. A shard's tags exist in the shard's document in the `shards` collection of the `config` database. To return all shards with a specific tag, use a sequence of operations that resemble the following, which will return only those shards tagged with `NYC`:

```
use config
db.shards.find({ tags: "NYC" })
```

You can find tag ranges for all *namespaces* in the `tags` collection of the `config` database. The output of `sh.status()` (page 349) displays all tag ranges. To return all shard key ranges tagged with `NYC`, use the following sequence of operations:

```
use config
db.tags.find({ tags: "NYC" })
```

Enforce Unique Keys for Sharded Collections

Overview

The `unique` constraint on indexes ensures that only one document can have a value for a field in a *collection*. For *sharded collections these unique indexes cannot enforce uniqueness* because insert and indexing operations are local to each shard. [10]

If your need to ensure that a field is always unique in all collections in a sharded environment, there are two options:

1. Enforce uniqueness of the *shard key* (page 296).

   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.

[10] If you specify a unique index on a sharded collection, MongoDB will be able to enforce uniqueness only among the documents located on a single shard *at the time of creation*.

2. Use a secondary collection to enforce uniqueness.

   Create a minimal collection that only contains the unique field and a reference to a document in the main collection. If you always insert into a secondary collection *before* inserting to the main collection, MongoDB will produce an error if you attempt to use a duplicate key.

   Note: If you have a small data set, you may not need to shard this collection and you can create multiple unique indexes. Otherwise you can shard on a single unique key.

Always use the default *acknowledged* (page 38) *write concern* (page 38) in conjunction with a *recent MongoDB driver* (page 419).

Unique Constraints on the Shard Key

Process To shard a collection using the `unique` constraint, specify the `shardCollection` (page 352) command in the following form:

```
db.runCommand( { shardCollection : "test.users" , key : { email : 1 } , unique : true } );
```

Remember that the `_id` field index is always unique. By default, MongoDB inserts an `ObjectId` into the `_id` field. However, you can manually insert your own value into the `_id` field and use this as the shard key. To use the `_id` field as the shard key, use the following operation:

```
db.runCommand( { shardCollection : "test.users" } )
```

Warning: In any sharded collection where you are *not* sharding by the `_id` field, you must ensure uniqueness of the `_id` field. The best way to ensure `_id` is always unique is to use `ObjectId`, or another universally unique identifier (UUID.)

Limitations

- You can only enforce uniqueness on one single field in the collection using this method.
- 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 305) and *query isolation* (page 306), as well as *high cardinality* (page 305). These ideal shard keys are not often the same keys that require uniqueness and requires a different approach.

Unique Constraints on Arbitrary Fields

If you cannot use a unique field as the shard key or if you need to enforce uniqueness over multiple fields, you must create another *collection* to act as a "proxy collection". This collection must contain both a reference to the original document (i.e. its `ObjectId`) and the unique key.

If you must shard this "proxy" collection, then shard on the unique key using the *above procedure* (page 339); otherwise, you can simply create multiple unique indexes on the collection.

Process Consider the following for the "proxy collection:"

```
{
  "_id" : ObjectId("...")
  "email" ": "..."
}
```

The `_id` field holds the `ObjectId` of the *document* it reflects, and the `email` field is the field on which you want to ensure uniqueness.

To shard this collection, use the following operation using the `email` field as the *shard key*:

```
db.runCommand( { shardCollection : "records.proxy" , key : { email : 1 } , unique : true } );
```

If you do not need to shard the proxy collection, use the following command to create a unique index on the `email` field:

```
db.proxy.ensureIndex( { "email" : 1 }, {unique : true} )
```

You may create multiple unique indexes on this collection if you do not plan to shard the `proxy` collection.

To insert documents, use the following procedure in the *JavaScript shell*:

```
use records;

var primary_id = ObjectId();

db.proxy.insert({
   "_id" : primary_id
   "email" : "example@example.net"
})

// if: the above operation returns successfully,
// then continue:

db.information.insert({
   "_id" : primary_id
   "email": "example@example.net"
   // additional information...
})
```

You must insert a document into the `proxy` collection first. If this operation succeeds, the `email` field is unique, and you may continue by inserting the actual document into the `information` collection.

See

The full documentation of: `db.collection.ensureIndex()` and `shardCollection` (page 352).

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.

Convert a Replica Set to a Replicated Sharded Cluster

Overview

Following this tutorial, you will convert a single 3-member replica set to a cluster that consists of 2 shards. Each shard will consist of an independent 3-member replica set.

The tutorial uses a test environment running on a local system UNIX-like system. You should feel encouraged to "follow along at home." If you need to perform this process in a production environment, notes throughout the document indicate procedural differences.

The procedure, from a high level, is as follows:

1. Create or select a 3-member replica set and insert some data into a collection.
2. Start the config databases and create a cluster with a single shard.
3. Create a second replica set with three new `mongod` instances.
4. Add the second replica set as a shard in the cluster.
5. Enable sharding on the desired collection or collections.

Process

Install MongoDB according to the instructions in the *MongoDB Installation Tutorial* (page 3).

Deploy a Replica Set with Test Data If have an existing MongoDB *replica set* deployment, you can omit the this step and continue from *Deploy Sharding Infrastructure* (page 342).

Use the following sequence of steps to configure and deploy a replica set and to insert test data.

1. Create the following directories for the first replica set instance, named `firstset`:

   - `/data/example/firstset1`
   - `/data/example/firstset2`
   - `/data/example/firstset3`

   To create directories, issue the following command:

   ```
   mkdir -p /data/example/firstset1 /data/example/firstset2 /data/example/firstset3
   ```

2. In a separate terminal window or GNU Screen window, start three `mongod` instances by running each of the following commands:

   ```
   mongod --dbpath /data/example/firstset1 --port 10001 --replSet firstset --oplogSize 700 --rest
   mongod --dbpath /data/example/firstset2 --port 10002 --replSet firstset --oplogSize 700 --rest
   mongod --dbpath /data/example/firstset3 --port 10003 --replSet firstset --oplogSize 700 --rest
   ```

   Note: The *`--oplogSize 700`* option restricts the size of the operation log (i.e. oplog) for each `mongod` instance to 700MB. Without the *`--oplogSize`* option, each `mongod` reserves approximately 5% of the free disk space on the volume. By limiting the size of the oplog, each instance starts more quickly. Omit this setting in production environments.

3. In a `mongo` shell session in a new terminal, connect to the mongodb instance on port 10001 by running the following command. If you are in a production environment, first read the note below.

```
mongo localhost:10001/admin
```

Note: Above and hereafter, if you are running in a production environment or are testing this process with `mongod` instances on multiple systems, replace "localhost" with a resolvable domain, hostname, or the IP address of your system.

4. In the `mongo` shell, initialize the first replica set by issuing the following command:

```
db.runCommand({"replSetInitiate" :
                    {"_id" : "firstset", "members" : [{"_id" : 1, "host" : "localhost:10001"},
                                                      {"_id" : 2, "host" : "localhost:10002"},
                                                      {"_id" : 3, "host" : "localhost:10003"}
             ]}})
{
        "info" : "Config now saved locally.  Should come online in about a minute.",
        "ok" : 1
}
```

5. In the `mongo` shell, create and populate a new collection by issuing the following sequence of JavaScript operations:

```
use test
switched to db test
people = ["Marc", "Bill", "George", "Eliot", "Matt", "Trey", "Tracy", "Greg", "Steve", "Kristina
for(var i=0; i<1000000; i++){
                              name = people[Math.floor(Math.random()*people.length)];
                              user_id = i;
                              boolean = [true, false][Math.floor(Math.random()*2)];
                              added_at = new Date();
                              number = Math.floor(Math.random()*10001);
                              db.test_collection.save({"name":name, "user_id":user_id, "boolean":
                             }
```

The above operations add one million documents to the collection `test_collection`. This can take several minutes, depending on your system.

The script adds the documents in the following form:

```
{ "_id" : ObjectId("4ed5420b8fc1dd1df5886f70"), "name" : "Greg", "user_id" : 4, "boolean" : true, "a
```

Deploy Sharding Infrastructure This procedure creates the three config databases that store the cluster's metadata.

Note: For development and testing environments, a single config database is sufficient. In production environments, use three config databases. Because config instances store only the *metadata* for the sharded cluster, they have minimal resource requirements.

1. Create the following data directories for three *config database* instances:

   - `/data/example/config1`
   - `/data/example/config2`
   - `/data/example/config3`

   Issue the following command at the system prompt:

```
mkdir -p /data/example/config1 /data/example/config2 /data/example/config3
```

2. In a separate terminal window or GNU Screen window, start the config databases by running the following commands:

```
mongod --configsvr --dbpath /data/example/config1 --port 20001
mongod --configsvr --dbpath /data/example/config2 --port 20002
mongod --configsvr --dbpath /data/example/config3 --port 20003
```

3. In a separate terminal window or GNU Screen window, start `mongos` instance by running the following command:

```
mongos --configdb localhost:20001,localhost:20002,localhost:20003 --port 27017 --chunkSize 1
```

Note: If you are using the collection created earlier or are just experimenting with sharding, you can use a small *`--chunkSize`* (1MB works well.) The default `chunkSize` of 64MB means that your cluster must have 64MB of data before the MongoDB's automatic sharding begins working.

In production environments, do not use a small shard size.

The `configdb` options specify the *configuration databases* (e.g. `localhost:20001`, `localhost:20002`, and `localhost:2003`). The `mongos` instance runs on the default "MongoDB" port (i.e. `27017`), while the databases themselves are running on ports in the `30001` series. In the this example, you may omit the *`--port 27017`* option, as `27017` is the default port.

4. Add the first shard in `mongos`. In a new terminal window or GNU Screen session, add the first shard, according to the following procedure:

   (a) Connect to the `mongos` with the following command:

   ```
   mongo localhost:27017/admin
   ```

   (b) Add the first shard to the cluster by issuing the `addShard` (page 351) command:

   ```
   db.runCommand( { addShard : "firstset/localhost:10001,localhost:10002,localhost:10003" } )
   ```

   (c) Observe the following message, which denotes success:

   ```
   { "shardAdded" : "firstset", "ok" : 1 }
   ```

Deploy a Second Replica Set This procedure deploys a second replica set. This closely mirrors the process used to establish the first replica set above, omitting the test data.

1. Create the following data directories for the members of the second replica set, named `secondset`:
   - `/data/example/secondset1`
   - `/data/example/secondset2`
   - `/data/example/secondset3`
2. In three new terminal windows, start three instances of `mongod` with the following commands:

```
mongod --dbpath /data/example/secondset1 --port 10004 --replSet secondset --oplogSize 700 --rest
mongod --dbpath /data/example/secondset2 --port 10005 --replSet secondset --oplogSize 700 --rest
mongod --dbpath /data/example/secondset3 --port 10006 --replSet secondset --oplogSize 700 --rest
```

Note: As above, the second replica set uses the smaller `oplogSize` configuration. Omit this setting in production environments.

---

3. In the `mongo` shell, connect to one mongodb instance by issuing the following command:

```
mongo localhost:10004/admin
```

4. In the `mongo` shell, initialize the second replica set by issuing the following command:

```
db.runCommand({"replSetInitiate" :
                     {"_id" : "secondset",
                      "members" : [{"_id" : 1, "host" : "localhost:10004"},
                                   {"_id" : 2, "host" : "localhost:10005"},
                                   {"_id" : 3, "host" : "localhost:10006"}
              ]}})

{
     "info" : "Config now saved locally.  Should come online in about a minute.",
     "ok" : 1
}
```

5. Add the second replica set to the cluster. Connect to the `mongos` instance created in the previous procedure and issue the following sequence of commands:

```
use admin
db.runCommand( { addShard : "secondset/localhost:10004,localhost:10005,localhost:10006" } )
```

This command returns the following success message:

```
{ "shardAdded" : "secondset", "ok" : 1 }
```

6. Verify that both shards are properly configured by running the `listShards` (page 352) command. View this and example output below:

```
db.runCommand({listShards:1})
{
       "shards" : [
              {
                     "_id" : "firstset",
                     "host" : "firstset/localhost:10001,localhost:10003,localhost:10002"
              },
              {
                     "_id" : "secondset",
                     "host" : "secondset/localhost:10004,localhost:10006,localhost:10005"
              }
       ],
     "ok" : 1
}
```

Enable Sharding MongoDB must have *sharding* enabled on *both* the database and collection levels.

Enabling Sharding on the Database Level Issue the `enableSharding` (page 352) command. The following example enables sharding on the "test" database:

```
db.runCommand( { enableSharding : "test" } )
{ "ok" : 1 }
```

Create an Index on the Shard Key MongoDB uses the shard key to distribute documents between shards. Once selected, you cannot change the shard key. Good shard keys:

- have values that are evenly distributed among all documents,

- group documents that are often accessed at the same time into contiguous chunks, and
- allow for effective distribution of activity among shards.

Typically shard keys are compound, comprising of some sort of hash and some sort of other primary key. Selecting a shard key depends on your data set, application architecture, and usage pattern, and is beyond the scope of this document. For the purposes of this example, we will shard the "number" key. This typically would *not* be a good shard key for production deployments.

Create the index with the following procedure:

```
use test
db.test_collection.ensureIndex({number:1})
```

See also:

The *Shard Key Overview* (page 296) and *Shard Key* (page 304) sections.

Shard the Collection Issue the following command:

```
use admin
db.runCommand( { shardCollection : "test.test_collection", key : {"number":1} })
{ "collectionsharded" : "test.test_collection", "ok" : 1 }
```

The collection `test_collection` is now sharded!

Over the next few minutes the Balancer begins to redistribute chunks of documents. You can confirm this activity by switching to the `test` database and running `db.stats()` or `db.printShardingStatus()`.

As clients insert additional documents into this collection, `mongos` distributes the documents evenly between the shards.

In the `mongo` shell, issue the following commands to return statics against each cluster:

```
use test
db.stats()
db.printShardingStatus()
```

Example output of the `db.stats()` command:

```
{
     "raw" : {
              "firstset/localhost:10001,localhost:10003,localhost:10002" : {
                       "db" : "test",
                       "collections" : 3,
                       "objects" : 973887,
                       "avgObjSize" : 100.33173458522396,
                       "dataSize" : 97711772,
                       "storageSize" : 141258752,
                       "numExtents" : 15,
                       "indexes" : 2,
                       "indexSize" : 56978544,
                       "fileSize" : 1006632960,
                       "nsSizeMB" : 16,
                       "ok" : 1
              },
              "secondset/localhost:10004,localhost:10006,localhost:10005" : {
                       "db" : "test",
                       "collections" : 3,
                       "objects" : 26125,
                       "avgObjSize" : 100.33286124401914,
```

```
                    "dataSize" : 2621196,
                    "storageSize" : 11194368,
                    "numExtents" : 8,
                    "indexes" : 2,
                    "indexSize" : 2093056,
                    "fileSize" : 201326592,
                    "nsSizeMB" : 16,
                    "ok" : 1
            }
    },
    "objects" : 1000012,
    "avgObjSize" : 100.33176401883178,
    "dataSize" : 100332968,
    "storageSize" : 152453120,
    "numExtents" : 23,
    "indexes" : 4,
    "indexSize" : 59071600,
    "fileSize" : 1207959552,
    "ok" : 1
}
```

Example output of the `db.printShardingStatus()` command:

```
--- Sharding Status ---
sharding version: { "_id" : 1, "version" : 3 }
shards:
       {  "_id" : "firstset",  "host" : "firstset/localhost:10001,localhost:10003,localhost:10002" }
       {  "_id" : "secondset",  "host" : "secondset/localhost:10004,localhost:10006,localhost:10005"
databases:
       {  "_id" : "admin",  "partitioned" : false,  "primary" : "config" }
       {  "_id" : "test",  "partitioned" : true,  "primary" : "firstset" }
                test.test_collection chunks:
                                                secondset     5
                                                firstset      186

[...]
```

In a few moments you can run these commands for a second time to demonstrate that *chunks* are migrating from `firstset` to `secondset`.

When this procedure is complete, you will have converted a replica set into a cluster where each shard is itself a replica set.

8.3 Sharded Cluster Reference

Consider the following reference material relevant to sharded cluster use and administration.

- *Sharding Commands* (page 347)
- `http://docs.mongodb.org/manual/reference/config-database`
- `http://docs.mongodb.org/manual/reference/mongos`

8.3.1 Sharding Commands

JavaScript Methods

`sh.addShard(host)`

Parameters

- **host** (*string*) – Specify the hostname of a database instance or a replica set configuration.

Use this method to add a database instance or replica set to a *sharded cluster*. This method must be run on a `mongos` instance. The `host` parameter can be in any of the following forms:

```
[hostname]
[hostname]:[port]
[set]/[hostname]
[set]/[hostname],[hostname]:port
```

You can specify shards using the hostname, or a hostname and port combination if the shard is running on a non-standard port.

Warning: Do not use `localhost` for the hostname unless your *configuration server* is also running on `localhost`.

The optimal configuration is to deploy shards across *replica sets*. To add a shard on a replica set you must specify the name of the replica set and the hostname of at least one member of the replica set. You must specify at least one member of the set, but can specify all members in the set or another subset if desired. `sh.addShard()` (page 347) takes the following form:

If you specify additional hostnames, all must be members of the same replica set.

```
sh.addShard("set-name/seed-hostname")
```

Example

```
sh.addShard("repl0/mongodb3.example.net:27327")
```

The `sh.addShard()` (page 347) method is a helper for the `addShard` (page 351) command. The `addShard` (page 351) command has additional options which are not available with this helper.

See also:

- `addShard` (page 351)
- *Sharded Cluster Administration* (page 298)
- *Add Shards to a Cluster* (page 315)
- *Remove Shards from an Existing Sharded Cluster* (page 329)

`sh.enableSharding(database)`

Parameters

- **database** (*string*) – Specify a database name to shard.

Enables sharding on the specified database. This does not automatically shard any collections, but makes it possible to begin sharding collections using `sh.shardCollection()` (page 348).

See also:

`sh.shardCollection()` (page 348)

`sh.shardCollection(namespace, key, unique)`

Parameters

- **namespace** (*string*) – The *namespace* of the collection to shard.
- **key** (*document*) – A *document* containing a *shard key* that the sharding system uses to *partition* and distribute objects among the shards.
- **unique** (*boolean*) – When true, the `unique` option ensures that the underlying index enforces uniqueness so long as the unique index is a prefix of the shard key.

Shards the named collection, according to the specified *shard key*. Specify shard keys in the form of a *document*. Shard keys may refer to a single document field, or more typically several document fields to form a "compound shard key."

See

`Size of Sharded Collection`

`sh.splitFind(namespace, query)`

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.
- **query** – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Splits the chunk containing the document specified by the `query` at its median point, creating two roughly equal chunks. Use `sh.splitAt()` (page 348) to split a collection in a specific point.

In most circumstances, you should leave chunk splitting to the automated processes. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitFind()` (page 348).

`sh.splitAt(namespace, query)`

Parameters

- **namespace** (*string*) – Specify the namespace (i.e. "`<database>.<collection>`") of the sharded collection that contains the chunk to split.
- **query** (*document*) – Specify a query to identify a document in a specific chunk. Typically specify the *shard key* for a document as the query.

Splits the chunk containing the document specified by the `query` as if that document were at the "middle" of the collection, even if the specified document is not the actual median of the collection. Use this command to manually split chunks unevenly. Use the "`sh.splitFind()` (page 348)" function to split a chunk at the actual median.

In most circumstances, you should leave chunk splitting to the automated processes within MongoDB. However, when initially deploying a *sharded cluster* it is necessary to perform some measure of *pre-splitting* using manual methods including `sh.splitAt()` (page 348).

`sh.moveChunk(collection, query, destination)`

Parameters

- **collection** (*string*) – Specify the sharded collection containing the chunk to migrate.

- **query** – Specify a query to identify documents in a specific chunk. Typically specify the *shard key* for a document as the query.
- **destination** (*string*) – Specify the name of the shard that you wish to move the designated chunk to.

Moves the chunk containing the documents specified by the `query` to the shard described by `destination`.

This function provides a wrapper around the `moveChunk`. In most circumstances, allow the *balancer* to automatically migrate *chunks*, and avoid calling `sh.moveChunk()` (page 348) directly.

See also:

"`moveChunk`" and "*Sharding* (page 295)" for more information.

`sh.`**`setBalancerState`**(*state*)

Parameters

- **state** (*boolean*) – `true` enables the balancer if disabled, and `false` disables the balancer.

Enables or disables the *balancer*. Use `sh.getBalancerState()` to determine if the balancer is currently enabled or disabled and `sh.isBalancerRunning()` (page 349) to check its current state.

See also:

- `sh.enableBalancing()`
- `sh.disableBalancing()`
- `sh.getBalancerHost()`
- `sh.getBalancerState()`
- `sh.isBalancerRunning()` (page 349)
- `sh.startBalancer()`
- `sh.stopBalancer()`
- `sh.waitForBalancer()`
- `sh.waitForBalancerOff()`

`sh.`**`isBalancerRunning`**()

Returns boolean

Returns true if the *balancer* process is currently running and migrating chunks and false if the balancer process is not running. Use `sh.getBalancerState()` to determine if the balancer is enabled or disabled.

See also:

- `sh.enableBalancing()`
- `sh.disableBalancing()`
- `sh.getBalancerHost()`
- `sh.getBalancerState()`
- `sh.setBalancerState()` (page 349)
- `sh.startBalancer()`
- `sh.stopBalancer()`
- `sh.waitForBalancer()`
- `sh.waitForBalancerOff()`

sh.**status**()

Returns A formatted report of the status of the *sharded cluster*, including data regarding the distribution of chunks.

sh.**addShardTag**(*shard*, *tag*)

New in version 2.2.

Parameters

- **shard** (*string*) – Specifies the name of the shard that you want to give a specific tag.
- **tag** (*string*) – Specifies the name of the tag that you want to add to the shard.

`sh.addShardTag()` (page 350) associates a shard with a tag or identifier. MongoDB uses these identifiers to direct *chunks* that fall within a tagged range to specific shards.

`sh.addTagRange()` (page 350) associates chunk ranges with tag ranges.

Always issue `sh.addShardTag()` (page 350) when connected to a `mongos` instance.

Example

The following example adds three tags, `NYC`, `LAX`, and `NRT`, to three shards:

```
sh.addShardTag("shard0000", "NYC")
sh.addShardTag("shard0001", "LAX")
sh.addShardTag("shard0002", "NRT")
```

See also:

- `sh.addTagRange()` (page 350) and
- `sh.removeShardTag()` (page 351)

sh.**addTagRange**(*namespace*, *minimum*, *maximum*, *tag*)

New in version 2.2.

Parameters

- **namespace** (*string*) – Specifies the namespace, in the form of `<database>.<collection>` of the sharded collection that you would like to tag.
- **minimum** (*document*) – Specifies the minimum value of the *shard key* range to include in the tag. Specify the minimum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.
- **maximum** (*document*) – Specifies the maximum value of the shard key range to include in the tag. Specify the maximum value in the form of `<fieldname>:<value>`. This value must be of the same BSON type or types as the shard key.
- **tag** (*string*) – Specifies the name of the tag to attach the range specified by the `minimum` and `maximum` arguments to.

`sh.addTagRange()` (page 350) attaches a range of values of the shard key to a shard tag created using the `sh.addShardTag()` (page 350) method. Use this operation to ensure that the documents that exist within the specified range exist on shards that have a matching tag.

Always issue `sh.addTagRange()` (page 350) when connected to a `mongos` instance.

Note: If you add a tag range to a collection using `sh.addTagRange()` (page 350), and then later drop the

collection or its database, MongoDB does not remove tag association. If you later create a new collection with the same name, the old tag association will apply to the new collection.

Example

Given a shard key of `{STATE:1,ZIP:1}`, create a tag range covering ZIP codes in New York State:

```
sh.addTagRange( "exampledb.collection",
        { STATE: "NY", ZIP: MinKey },
        { STATE:"NY", ZIP: MaxKey },
        "NY"
        )
```

See also:

`sh.addShardTag()` (page 350), `sh.removeShardTag()` (page 351)

`sh.`**`removeShardTag`**(*shard*, *tag*)

New in version 2.2.

Parameters

- **shard** (*string*) – Specifies the name of the shard that you want to remove a tag from.
- **tag** (*string*) – Specifies the name of the tag that you want to remove from the shard.

Removes the association between a tag and a shard.

Always issue `sh.removeShardTag()` (page 351) when connected to a `mongos` instance.

See also:

`sh.addShardTag()` (page 350), `sh.addTagRange()` (page 350)

`sh.`**`help`**`()`

Returns a basic help text for all sharding related shell functions.

Database Commands

The following database commands support *sharded clusters*.

`addShard`

Parameters

- **hostname** (*string*) – a hostname or replica-set/hostname string.
- **name** (*string*) – Optional. Unless specified, a name will be automatically provided to uniquely identify the shard.
- **maxSize** (*integer*) – Optional, megabytes. Limits the maximum size of a shard. If `maxSize` is `0` then MongoDB will not limit the size of the shard.

Use the `addShard` (page 351) command to add a database instance or replica set to a *sharded cluster*. You must run this command when connected a `mongos` instance.

The command takes the following form:

```
{ addShard: "<hostname><:port>" }
```

Example

```
db.runCommand({addShard: "mongodb0.example.net:27027"})
```

Replace <hostname><:port> with the hostname and port of the database instance you want to add as a shard.

Warning: Do not use localhost for the hostname unless your *configuration server* is also running on localhost.

The optimal configuration is to deploy shards across *replica sets*. To add a shard on a replica set you must specify the name of the replica set and the hostname of at least one member of the replica set. You must specify at least one member of the set, but can specify all members in the set or another subset if desired. addShard (page 351) takes the following form:

```
{ addShard: "replica-set/hostname:port" }
```

Example

```
db.runCommand( { addShard: "repl0/mongodb3.example.net:27327"} )
```

If you specify additional hostnames, all must be members of the same replica set.

Send this command to only one mongos instance, it will store shard configuration information in the *config database*.

Note: Specify a maxSize when you have machines with different disk capacities, or if you want to limit the amount of data on some shards.

The maxSize constraint prevents the *balancer* from migrating chunks to the shard when the value of mem.mapped exceeds the value of maxSize.

See also:

- sh.addShard() (page 347)
- *Sharded Cluster Administration* (page 298)
- *Add Shards to a Cluster* (page 315)
- *Remove Shards from an Existing Sharded Cluster* (page 329)

listShards

Use the listShards (page 352) command to return a list of configured shards. The command takes the following form:

```
{ listShards: 1 }
```

enableSharding

The enableSharding (page 352) command enables sharding on a per-database level. Use the following command form:

```
{ enableSharding: "<database name>" }
```

Once you've enabled sharding in a database, you can use the shardCollection (page 352) command to begin the process of distributing data among the shards.

shardCollection

The shardCollection (page 352) command marks a collection for sharding and will allow data to begin

distributing among shards. You must run enableSharding (page 352) on a database before running the shardCollection (page 352) command.

```
{ shardCollection: "<db>.<collection>", key: <shardkey> }
```

This enables sharding for the collection specified by <collection> in the database named <db>, using the key <shardkey> to distribute documents among the shard. <shardkey> is a document, and takes the same form as an *index specification document* (page 52).

Choosing the right shard key to effectively distribute load among your shards requires some planning.

See also:

Sharding (page 295) for more information related to sharding. Also consider the section on *Shard Key Selection* (page 296) for documentation regarding shard keys.

Warning: There's no easy way to disable sharding after running shardCollection (page 352). In addition, you cannot change shard keys once set. If you must convert a sharded cluster to a *standalone* node or *replica set*, you must make a single backup of the entire cluster and then restore the backup to the standalone mongod or the replica set..

shardingState

shardingState (page 353) is an admin command that reports if mongod is a member of a *sharded cluster*. shardingState (page 353) has the following prototype form:

```
{ shardingState: 1 }
```

For shardingState (page 353) to detect that a mongod is a member of a sharded cluster, the mongod must satisfy the following conditions:

1. the mongod is a primary member of a replica set, and
2. the mongod instance is a member of a sharded cluster.

If shardingState (page 353) detects that a mongod is a member of a sharded cluster, shardingState (page 353) returns a document that resembles the following prototype:

```
{
  "enabled" : true,
  "configServer" : "<configdb-string>",
  "shardName" : "<string>",
  "shardHost" : "string:",
  "versions" : {
       "<database>.<collection>" : Timestamp(<...>),
       "<database>.<collection>" : Timestamp(<...>)
  },
  "ok" : 1
}
```

Otherwise, shardingState (page 353) will return the following document:

```
{ "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
```

The response from shardingState (page 353) when used with a *config server* is:

```
{ "enabled": false, "ok": 1 }
```

Note: mongos instances do not provide the shardingState (page 353).

> **Warning:** This command obtains a write lock on the affected database and will block other operations until it has completed; however, the operation is typically short lived.

removeShard

Starts the process of removing a shard from a *cluster*. This is a multi-stage process. Begin by issuing the following command:

```
{ removeShard : "[shardName]" }
```

The balancer will then migrate chunks from the shard specified by `[shardName]`. This process happens slowly to avoid placing undue load on the overall cluster.

The command returns immediately, with the following message:

```
{ msg : "draining started successfully" , state: "started" , shard: "shardName" , ok : 1 }
```

If you run the command again, you'll see the following progress output:

```
{ msg: "draining ongoing" , state: "ongoing" , remaining: { chunks: 23 , dbs: 1 }, ok: 1 }
```

The `remaining` *document* specifies how many chunks and databases remain on the shard. Use `db.printShardingStatus()` to list the databases that you must move from the shard.

Each database in a sharded cluster has a primary shard. If the shard you want to remove is also the primary of one of the cluster's databases, then you must manually move the database to a new shard. This can be only after the shard is empty. See the `movePrimary` command for details.

After removing all chunks and databases from the shard, you may issue the command again, to return:

```
{ msg: "remove shard completed successfully", state: "completed", host: "shardName", ok : 1 }
```

CHAPTER 9

Frequently Asked Questions

9.1 FAQ: MongoDB Fundamentals

This document addresses basic high level questions about MongoDB and it's use.

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 355) or post your question to the MongoDB User Mailing List[1].

Frequently Asked Questions:

9.1.1 What kind of database is MongoDB?

MongoDB is *document*-oriented DBMS. Think of MySQL but with *JSON*-like objects comprising the data model, rather than RDBMS tables. Significantly, MongoDB supports neither joins nor transactions. However, it features secondary indexes, an expressive query language, atomic writes on a per-document level, and fully-consistent reads.

Operationally, MongoDB features master-slave replication with automated failover and built-in horizontal scaling via automated range-based partitioning.

Note: MongoDB uses *BSON*, a binary object format similar to, but more expressive than *JSON*.

[1] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

9.1.2 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

`http://docs.mongodb.org/manual/reference/sql-comparison`

9.1.3 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

`http://docs.mongodb.org/manual/reference/sql-comparison`

9.1.4 What languages can I use to work with MongoDB?

MongoDB *client drivers* exist for all of the most popular programming languages, and many other ones. See the latest list of drivers[2] for details.

See also:

"`http://docs.mongodb.org/manual/applications/drivers`."

9.1.5 Does MongoDB support SQL?

No.

However, MongoDB does support a rich, ad-hoc query language of its own.

See also:

The `http://docs.mongodb.org/manual/reference/operators` and the `Query Overview` pages.

9.1.6 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 app, gaming, e-commerce, analytics, archiving, and logging.

[2] http://docs.mongodb.org/ecosystem/drivers

Do not use MongoDB for systems that require SQL, joins, and multi-object transactions.

9.1.7 Does MongoDB support transactions?

MongoDB does not provide ACID transactions.

However, MongoDB does provide some basic transactional capabilities. Atomic operations are possible within the scope of a single document: that is, we can debit `a` and credit `b` as a transaction if they are fields within the same document. Because documents can be rich, some documents contain thousands of fields, with support for testing fields in sub-documents.

Additionally, you can make writes in MongoDB durable (the 'D' in ACID). To get durable writes, you must enable journaling, which is on by default in 64-bit builds. You must also issue writes with a write concern of `{j: true}` to ensure that the writes block until the journal has synced to disk.

Users have built successful e-commerce systems using MongoDB, but applications requiring multi-object commits with rollback generally aren't feasible.

9.1.8 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 it's usage is dynamic. If another process suddenly needs half the server's RAM, MongoDB will yield cached memory to the other process.

Technically, the operating system's virtual memory subsystem manages MongoDB's memory. This means that MongoDB will use as much free memory as it can, swapping to disk as needed. Deployments with enough memory to fit the application's working data set in RAM will achieve the best performance.

See also:

FAQ: MongoDB Diagnostics (page 389) for answers to additional questions about MongoDB and Memory use.

9.1.9 How do I configure the cache size?

MongoDB has no configurable cache. MongoDB uses all *free* memory on the system automatically by way of memory-mapped files. Operating systems use the same approach with their file system caches.

9.1.10 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.

9.1.11 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.

9.1.12 Are writes written to disk immediately, or lazily?

Writes are physically written to the *journal* (page 100) within 100 milliseconds, by default. At that point, the write is "durable" in the sense that after a pull-plug-from-wall event, the data will still be recoverable after a hard restart. See `journalCommitInterval` for more information on the journal commit window.

While the journal commit is nearly instant, MongoDB writes to the data files lazily. MongoDB may wait to write data to the data files for as much as one minute by default. This does not affect durability, as the journal has enough information to ensure crash recovery. To change the interval for writing to the data files, see `syncdelay`.

9.1.13 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.

9.1.14 What are the limitations of 32-bit versions of MongoDB?

MongoDB uses *memory-mapped files* (page 384). When running a 32-bit build of MongoDB, the total storage size for the server, including data and indexes, is 2 gigabytes. For this reason, do not deploy MongoDB to production on 32-bit machines.

If you're running a 64-bit build of MongoDB, there's virtually no limit to storage size. For production deployments, 64-bit builds and operating systems are strongly recommended.

See also:

"Blog Post: 32-bit Limitations[3]"

Note: 32-bit builds disable *journaling* by default because journaling further limits the maximum amount of data that the database can store.

9.2 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 355) or post your question to the MongoDB User Mailing List[4].

[3] http://blog.mongodb.org/post/137788967/32-bit-limitations
[4] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

Frequently Asked Questions:

9.2.1 What is a namespace in MongoDB?

A "namespace" is the concatenation of the *database* name and the *collection* names [5] with a period character in between.

Collections are containers for documents that share one or more indexes. Databases are groups of collections stored on disk using a single set of data files. [6]

For an example `acme.users` namespace, `acme` is the database name and `users` is the collection name. Period characters **can** occur in collection names, so that `acme.user.history` is a valid namespace, with `acme` as the database name, and `user.history` as the collection name.

While data models like this appear to support nested collections, the collection namespace is flat, and there is no difference from the perspective of MongoDB between `acme`, `acme.users`, and `acme.records`.

9.2.2 How do you copy all objects from one collection to another?

In the `mongo` shell, you can use the following operation to duplicate the entire collection:

```
db.source.copyTo(newCollection)
```

[5] Each index also has its own namespace.
[6] MongoDB database have an configurable limit on the `number of namespaces` in a database.

Warning: When using `db.collection.copyTo()` check field types to ensure that the operation does not remove type information from documents during the translation from *BSON* to *JSON*. Consider using `cloneCollection()` to maintain type fidelity.

Also consider the `cloneCollection` *command* that may provide some of this functionality.

9.2.3 If you remove a document, does MongoDB remove it from disk?

Yes.

When you use `db.collection.remove()`, the object will no longer exist in MongoDB's on-disk data storage.

9.2.4 When does MongoDB write updates to disk?

MongoDB flushes writes to disk on a regular interval. In the default configuration, MongoDB writes data to the main data files on disk every 60 seconds and commits the *journal* roughly every 100 milliseconds. These values are configurable with the `journalCommitInterval` and `syncdelay`.

These values represent the *maximum* amount of time between the completion of a write operation and the point when the write is durable in the journal, if enabled, and when MongoDB flushes data to the disk. In many cases MongoDB and the operating system flush data to disk more frequently, so that the above values represents a theoretical maximum.

However, by default, MongoDB uses a "lazy" strategy to write to disk. This is advantageous in situations where the database receives a thousand increments to an object within one second, MongoDB only needs to flush this data to disk once. In addition to the aforementioned configuration options, you can also use `fsync` and `getLastError` to modify this strategy.

9.2.5 How do I do transactions and locking in MongoDB?

MongoDB does not have support for traditional locking or complex transactions with rollback. MongoDB aims to be lightweight, fast, and predictable in its performance. This is similar to the MySQL MyISAM autocommit model. By keeping transaction support extremely simple, MongoDB can provide greater performance especially for *partitioned* or *replicated* systems with a number of database server processes.

MongoDB *does* have support for atomic operations *within* a single document. Given the possibilities provided by nested documents, this feature provides support for a large number of use-cases.

See also:

The `http://docs.mongodb.org/manual/tutorial/isolate-sequence-of-operations` page.

9.2.6 How do you aggregate data with MongoDB?

In version 2.1 and later, you can use the new "*aggregation framework* (page 151)," with the `aggregate` command.

MongoDB also supports *map-reduce* with the `mapReduce` command, as well as basic aggregation with the `group`, `count`, and `distinct`. commands.

See also:

The *Aggregation* (page 151) page.

9.2.7 Why does MongoDB log so many "Connection Accepted" events?

If you see a very large number connection and re-connection messages in your MongoDB log, then clients are frequently connecting and disconnecting to the MongoDB server. This is normal behavior for applications that do not use request pooling, such as CGI. Consider using FastCGI, an Apache Module, or some other kind of persistent application server to decrease the connection overhead.

If these connections do not impact your performance you can use the run-time `quiet` option or the command-line option `--quiet` to suppress these messages from the log.

9.2.8 Does MongoDB run on Amazon EBS?

Yes.

MongoDB users of all sizes have had a great deal of success using MongoDB on the EC2 platform using EBS disks.

See also:

Amazon EC2[7]

9.2.9 Why are MongoDB's data files so large?

MongoDB aggressively preallocates data files to reserve space and avoid file system fragmentation. You can use the `smallfiles` setting to modify the file preallocation strategy.

See also:

Why are the files in my data directory larger than the data in my database? (page 385)

9.2.10 How do I optimize storage use for small documents?

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the `_id` field explicitly.

  MongoDB clients automatically add an `_id` field to each document and generate a unique 12-byte *ObjectId* for the `_id` field. Furthermore, MongoDB always indexes the `_id` field. For smaller documents this may account for a significant amount of space.

  To optimize storage use, users can specify a value for the `_id` field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the `_id` field that would have occupied space in another portion of the document.

  You can store any value in the `_id` field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field's value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.

  MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of documents that resemble the following:

[7] http://docs.mongodb.org/ecosystem/platforms/amazon-ec2

```
{ last_name : "Smith", best_score: 3.9 }
```

If you shorten the filed named `last_name` to `lname` and the field name `best_score` to `score`, as follows, you could save 9 bytes per document.

```
{ lname : "Smith", score : 3.9 }
```

Shortening field names reduces expressiveness and does not provide considerable benefit on for larger documents and where document overhead is not significant concern. Shorter field names do not reduce the size of indexes, because indexes have a predefined structure.

In general it is not necessary to use short field names.

- Embed documents.

  In some cases you may want to embed documents in other documents and save on the per-document overhead.

9.2.11 When should I use GridFS?

For documents in a MongoDB collection, you should always use *GridFS* for storing files larger than 16 MB.

In some situations, storing large files may be more efficient in a MongoDB database than on a system-level filesystem.

- If your filesystem limits the number of files in a directory, you can use GridFS to store as many files as needed.
- When you want to keep your files and metadata automatically synced and deployed across a number of systems and facilities. When using *geographically distributed replica sets* (page 239) 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 58).

9.2.12 How does MongoDB address SQL or Query injection?

BSON

As a client program assembles a query in MongoDB, it builds a BSON object, not a string. Thus traditional SQL injection attacks are not a problem. More details and some nuances are covered below.

MongoDB represents queries as *BSON* objects. Typically `client libraries` provide a convenient, injection free, process to build these objects. Consider the following C++ example:

```
BSONObj my_query = BSON( "name" << a_name );
auto_ptr<DBClientCursor> cursor = c.query("tutorial.persons", my_query);
```

Here, `my_query` then will have a value such as `{ name : "Joe" }`. If `my_query` contained special characters, for example `,`, `:`, and `{`, the query simply wouldn't match any documents. For example, users cannot hijack a query and convert it to a delete.

JavaScript

Note: You can disable all server-side execution of JavaScript, by passing the `--noscripting` option on the command line or setting `noscripting` in a configuration file.

All of the following MongoDB operations permit you to run arbitrary JavaScript expressions directly on the server:

- `$where`
- `db.eval()`
- `mapReduce`
- `group`

You must exercise care in these cases to prevent users from submitting malicious JavaScript.

Fortunately, you can express most queries in MongoDB without JavaScript and for queries that require JavaScript, you can mix JavaScript and non-JavaScript in a single query. Place all the user-supplied fields directly in a *BSON* field and pass JavaScript code to the `$where` field.

- If you need to pass user-supplied values in a `$where` clause, you may escape these values with the `CodeWScope` mechanism. When you set user-submitted values as variables in the scope document, you can avoid evaluating them on the database server.
- If you need to use `db.eval()` with user supplied values, you can either use a `CodeWScope` or you can supply extra arguments to your function. For instance:

  ```
  db.eval(function(userVal){...},
          user_value);
  ```

  This will ensure that your application sends `user_value` to the database server as data rather than code.

Dollar Sign Operator Escaping

Field names in MongoDB's query language have semantic meaning. The dollar sign (i.e `$`) is a reserved character used to represent `operators` (i.e. `$inc`.) Thus, you should ensure that your application's users cannot inject operators into their inputs.

In some cases, you may wish to build a BSON object with a user-provided key. In these situations, keys will need to substitute the reserved `$` and `.` characters. Any character is sufficient, but consider using the Unicode full width equivalents: `U+FF04` (i.e. "$") and `U+FF0E` (i.e. ".").

Consider the following example:

```
BSONObj my_object = BSON( a_key << a_name );
```

The user may have supplied a `$` value in the `a_key` value. At the same time, `my_object` might be `{ $where : "things" }`. Consider the following cases:

- **Insert**. Inserting this into the database does no harm. The insert process does not evaluate the object as a query.

  Note: MongoDB client drivers, if properly implemented, check for reserved characters in keys on inserts.

- **Update**. The `db.collection.update()` operation permits `$` operators in the update argument but does not support the `$where` operator. Still, some users may be able to inject operators that can manipulate a single document only. Therefore your application should escape keys, as mentioned above, if reserved characters are possible.

- **Query** Generally this is not a problem for queries that resemble `{ x : user_obj }`: dollar signs are not top level and have no effect. Theoretically it may be possible for the user to build a query themselves. But checking the user-submitted content for `$` characters in key names may help protect against this kind of injection.

Driver-Specific Issues

See the "PHP MongoDB Driver Security Notes[8]" page in the PHP driver documentation for more information

9.2.13 How does MongoDB provide concurrency?

MongoDB implements a readers-writer lock. This means that at any one time, only one client may be writing or any number of clients may be reading, but that reading and writing cannot occur simultaneously.

In standalone and *replica sets* the lock's scope applies to a single `mongod` instance or *primary* instance. In a sharded cluster, locks apply to each individual shard, not to the whole cluster.

For more information, see *FAQ: Concurrency* (page 370).

9.2.14 What is the compare order for BSON types?

MongoDB permits documents within a single collection to have fields with different *BSON* types. For instance, the following documents may exist within a single collection.

```
{ x: "string" }
{ x: 42 }
```

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

Consider the following `mongo` example:

[8] http://us.php.net/manual/en/mongo.security.php

```
db.test.insert( {x : 3 } );
db.test.insert( {x : 2.9 } );
db.test.insert( {x : new Date() } );
db.test.insert( {x : true } );

db.test.find().sort({x:1});
{ "_id" : ObjectId("4b03155dce8de6586fb002c7"), "x" : 2.9 }
{ "_id" : ObjectId("4b03154cce8de6586fb002c6"), "x" : 3 }
{ "_id" : ObjectId("4b031566ce8de6586fb002c9"), "x" : true }
{ "_id" : ObjectId("4b031563ce8de6586fb002c8"), "x" : "Tue Nov 17 2009 16:28:03 GMT-0500 (EST)" }
```

The `$type` operator provides access to *BSON type* comparison in the MongoDB query syntax. See the documentation on *BSON types* and the `$type` operator for additional information.

> **Warning:** Storing values of the different types in the same field in a collection is *strongly* discouraged.

See also:

- The `Tailable Cursors` page for an example of a C++ use of `MinKey`.

9.2.15 How do I query for fields that have null values?

Fields in a document may store `null` values, as in a notional collection, `test`, with the following documents:

```
{ _id: 1, cancelDate: null }
{ _id: 2 }
```

Different query operators treat `null` values differently:

- The `{ cancelDate : null }` query matches documents that either contains the `cancelDate` field whose value is `null` *or* that do not contain the `cancelDate` field:

  ```
  db.test.find( { cancelDate: null } )
  ```

  The query returns both documents:

  ```
  { "_id" : 1, "cancelDate" : null }
  { "_id" : 2 }
  ```

- The `{ cancelDate : { $type: 10 } }` query matches documents that contains the `cancelDate` field whose value is `null` *only*; i.e. the value of the `cancelDate` field is of BSON Type `Null` (i.e. `10`) :

  ```
  db.test.find( { cancelDate : { $type: 10 } } )
  ```

  The query returns only the document that contains the `null` value:

  ```
  { "_id" : 1, "cancelDate" : null }
  ```

- The `{ cancelDate : { $exists: false } }` query matches documents that do not contain the `cancelDate` field:

  ```
  db.test.find( { cancelDate : { $exists: false } } )
  ```

  The query returns only the document that does *not* contain the `cancelDate` field:

  ```
  { "_id" : 2 }
  ```

See also:

The reference documentation for the `$type` and `$exists` operators.

9.2.16 Are there any restrictions on the names of Collections?

Collection names can be any UTF-8 string with the following exceptions:

- A collection name should begin with a letter or an underscore.
- The empty string (`""`) is not a valid collection name.
- Collection names cannot contain the `$` character. (version 2.2 only)
- Collection names cannot contain the null character: `\0`
- Do not name a collection using the `system.` prefix. MongoDB reserves `system.` for system collections, such as the `system.indexes` collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

If your collection name includes special characters, such as the underscore character, then to access the collection use the `db.getCollection()` method or a similar method for your driver[9].

Example

To create a collection `_foo` and insert the `{ a : 1 }` document, use the following operation:

```
db.getCollection("_foo").insert( { a : 1 } )
```

To perform a query, use the `find()` method, in as the following:

```
db.getCollection("_foo").find()
```

9.2.17 How do I isolate cursors from intervening write operations?

MongoDB cursors can return the same document more than once in some situations. [10] You can use the `snapshot()` method on a cursor to isolate the operation for a very specific case.

`snapshot()` traverses the index on the `_id` field and guarantees that the query will return each document (with respect to the value of the `_id` field) no more than once. [11]

The `snapshot()` does not guarantee that the data returned by the query will reflect a single moment in time *nor* does it provide isolation from insert or delete operations.

Warning:
- You **cannot** use `snapshot()` with *sharded collections*.
- You **cannot** use `snapshot()` with `sort()` or `hint()` cursor methods.

As an alternative, if your collection has a field or fields that are never modified, you can use a *unique* index on this field or these fields to achieve a similar result as the `snapshot()`. Query with `hint()` to explicitly force the query to use that index.

[9] http://api.mongodb.org/

[10] As a cursor returns documents other operations may interleave with the query: if some of these operations are *updates* (page 77) that cause the document to move (in the case of a table scan, caused by document growth,) or that change the indexed field on the index used by the query; then the cursor will return the same document more than once.

[11] MongoDB does not permit changes to the value of the `_id` field; it is not possible for a cursor that transverses this index to pass the same document more than once.

9.2.18 When should I embed documents within other documents?

When *modeling data in MongoDB* (page 43), embedding is frequently the choice for:

- "contains" relationships between entities.
- one-to-many relationships when the "many" objects *always* appear with or are viewed in the context of their parents.

You should also consider embedding for performance reasons if you have a collection with a large number of small documents. Nevertheless, if small, separate documents represent the natural model for the data, then you should maintain that model.

If, however, you can group these small documents by some logical relationship *and* you frequently retrieve the documents by this grouping, you might consider "rolling-up" the small documents into larger documents that contain an array of subdocuments. Keep in mind that if you often only need to retrieve a subset of the documents within the group, then "rolling-up" the documents may not provide better performance.

"Rolling up" these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses.

Additionally, "rolling up" documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields *and* there would be fewer associated key entries in the corresponding index. See *Indexing Overview* (page 185) for more information on indexes.

9.2.19 Can I manually pad documents to prevent moves during updates?

An update can cause a document to move on disk if the document grows in size. To *minimize* document movements, MongoDB uses *padding* (page 41).

You should not have to pad manually because MongoDB adds *padding automatically* (page 41) and can adaptively adjust the amount of padding added to documents to prevent document relocations following updates.

You can change the default `paddingFactor` calculation by using the the `collMod` command with the `usePowerOf2Sizes` flag. The `usePowerOf2Sizes` flag ensures that MongoDB allocates document space in sizes that are powers of 2, which helps ensure that MongoDB can efficiently reuse free space created by document deletion or relocation.

However, in those exceptions where you must pad manually, you can use the strategy of first adding a temporary field to a document and then `$unset` the field, as in the following example:

```
var myTempPadding = [ "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                      "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"];

db.myCollection.insert( { _id: 5, paddingField: myTempPadding } );

db.myCollection.update( { _id: 5 },
                        { $unset: { paddingField: "" } }
                      )

db.myCollection.update( { _id: 5 },
                        { $set: { realField: "Some text that I might have needed padding for" } }
                      )
```

See also:

Padding Factor (page 41)

9.3 FAQ: The mongo Shell

9.3.1 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
...
...
>
```

9.3.2 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();
```

9.3.3 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*

9.3.4 How can I customize the mongo shell prompt?

New in version 1.9.

You can change the `mongo` shell prompt by setting the `prompt` variable. This makes it possible to display additional information in the prompt.

Set `prompt` to any string or arbitrary JavaScript code that returns a string, consider the following examples:

- Set the shell prompt to display the hostname and the database issued:

  ```
  var host = db.serverStatus().host;
  var prompt = function() { return db+"@"+host+"> "; }
  ```

  The `mongo` shell prompt should now reflect the new prompt:

  ```
  test@my-machine.local>
  ```

- Set the shell prompt to display the database statistics:

  ```
  var prompt = function() {
                  return "Uptime:"+db.serverStatus().uptime+" Documents:"+db.stats().objects+" > "
               }
  ```

  The `mongo` shell prompt should now reflect the new prompt:

  ```
  Uptime:1052 Documents:25024787 >
  ```

You can add the logic for the prompt in the *.mongorc.js* file to set the prompt each time you start up the `mongo` shell.

9.3.5 Can I edit long shell operations with an external text editor?

You can use your own editor in the `mongo` shell by setting the `EDITOR` environment variable before starting the `mongo` shell. Once in the `mongo` shell, you can edit with the specified editor by typing `edit <variable>` or `edit <function>`, as in the following example:

1. Set the `EDITOR` variable from the command line prompt:

   ```
   EDITOR=vim
   ```

2. Start the `mongo` shell:

   ```
   mongo
   ```

3. Define a function `myFunction`:

   ```
   function myFunction () { }
   ```

4. Edit the function using your editor:

   ```
   edit myFunction
   ```

   The command should open the `vim` edit session. Remember to save your changes.

5. Type `myFunction` to see the function definition:

   ```
   myFunction
   ```

   The result should be the changes from your saved edit:

   ```
   function myFunction() {
       print("This was edited");
   }
   ```

9.4 FAQ: Concurrency

Changed in version 2.2.

MongoDB allows multiple clients to read and write a single corpus of data using a locking system to ensure that all clients receive the same view of the data *and* to prevent multiple applications from modifying the exact same pieces of data at the same time. Locks help guarantee that all writes to a single document occur either in full or not at all.

See also:

Presentation on Concurrency and Internals in 2.2[12]

9.4.1 What type of locking does MongoDB use?

MongoDB uses a readers-writer [13] lock that allows concurrent reads access to a database but gives exclusive access to a single write operation.

When a read lock exists, many read operations may use this lock. However, when a write lock exists, a single write operation holds the lock exclusively, and no other read *or* write operations may share the lock.

Locks are "writer greedy," which means writes have preference over reads. When both a read and write are waiting for a lock, MongoDB grants the lock to the write.

9.4.2 How granular are locks in MongoDB?

Changed in version 2.2.

Beginning with version 2.2, MongoDB implements locks on a per-database basis for most read and write operations. Some global operations, typically short lived operations involving multiple databases, still require a global "instance" wide lock. Before 2.2, there is only one "global" lock per `mongod` instance.

For example, if you have six databases and one takes a write lock, the other five are still available for read and write.

9.4.3 How do I see the status of locks on my `mongod` instances?

For reporting on lock utilization information on locks, use any of the following methods:

- `db.serverStatus()`,
- `db.currentOp()`,
- `mongotop`,
- `mongostat`, and/or
- the MongoDB Management Service (MMS)[14]

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()`.

[12] http://www.mongodb.com/presentations/concurrency-internals-mongodb-2-2

[13] You may be familiar with a "readers-writer" lock as "multi-reader" or "shared exclusive" lock. See the Wikipedia page on Readers-Writer Locks (http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock) for more information.

[14] http://mms.mongodb.com/

9.4.4 Does a read or write operation ever yield the lock?

New in version 2.0.

Read and write operations will yield their locks if the `mongod` receives a *page fault or* fetches data that is unlikely to be in memory. Yielding allows other operations that only need to access documents that are already in memory to complete while `mongod` loads documents into memory.

Additionally, write operations that affect multiple documents (i.e. `update()` with the `multi` parameter,) will yield periodically to allow read operations during these long write operations. Similarly, long running read locks will yield periodically to ensure that write operations have the opportunity to complete.

Changed in version 2.2: The use of yielding expanded greatly in MongoDB 2.2. Including the "yield for page fault." MongoDB tracks the contents of memory and predicts whether data is available before performing a read. If MongoDB predicts that the data is not in memory a read operation yields its lock while MongoDB loads the data to memory. Once data is available in memory, the read will reacquire the lock to complete the operation.

9.4.5 Which operations lock the database?

Changed in version 2.2.

The following table lists common database operations and the types of locks they use:

Operation	Lock Type
Issue a query	Read lock
Get more data from a *cursor*	Read lock
Insert data	Write lock
Remove data	Write lock
Update data	Write lock
Map-reduce	Read lock and write lock, unless operations are specified as non-atomic. Portions of map-reduce jobs can run concurrently.
Create an index	Building an index in the foreground, which is the default, locks the database for extended periods of time.
`db.eval()`	Write lock. `db.eval()` blocks all other JavaScript processes.
`eval`	Write lock. If used with the `nolock` lock option, the `eval` option does not take a write lock and cannot write data to the database.
`aggregate()`	Read lock

9.4.6 Which administrative commands lock the database?

Certain administrative commands can exclusively lock the database for extended periods of time. In some deployments, for large databases, you may consider taking the `mongod` instance offline so that clients are not affected. For example, if a `mongod` is part of a *replica set*, take the `mongod` offline and let other members of the set service load while maintenance is in progress.

The following administrative operations require an exclusive (i.e. write) lock on the database for extended periods:

- `db.collection.ensureIndex()`, when issued *without* setting `background` to `true`,
- `reIndex`,
- `compact`,
- `db.repairDatabase()`,
- `db.createCollection()`, when creating a very large (i.e. many gigabytes) capped collection,

- db.collection.validate(), and
- db.copyDatabase(). This operation may lock all databases. See *Does a MongoDB operation ever lock more than one database?* (page 372).

The db.collection.group() operation takes a read lock and does not allow any other threads to execute JavaScript while it is running.

The following administrative commands lock the database but only hold the lock for a very short time:

- db.collection.dropIndex(),
- db.getLastError(),
- db.isMaster() (page 286),
- rs.status() (page 286) (i.e. replSetGetStatus (page 291),)
- db.serverStatus(),
- db.auth(), and
- db.addUser().

9.4.7 Does a MongoDB operation ever lock more than one database?

The following MongoDB operations lock multiple databases:

- db.copyDatabase() must lock the entire mongod instance at once.
- *Journaling*, which is an internal operation, locks all databases for short intervals. All databases share a single journal.
- *User authentication* (page 136) locks the admin database as well as the database the user is accessing.
- All writes to a replica set's *primary* lock both the database receiving the writes and the local database. The lock for the local database allows the mongod to write to the primary's *oplog*.

9.4.8 How does sharding affect concurrency?

Sharding improves concurrency by distributing collections over multiple mongod instances, allowing shard servers (i.e. mongos processes) to perform any number of operations concurrently to the various downstream mongod instances.

Each mongod instance is independent of the others in the shard cluster and uses the MongoDB *readers-writer lock* (page 370)). The operations on one mongod instance do not block the operations on any others.

9.4.9 How does concurrency affect a replica set primary?

In *replication*, when MongoDB writes to a collection on the *primary*, MongoDB also writes to the primary's *oplog*, which is a special collection in the local database. Therefore, MongoDB must lock both the collection's database and the local database. The mongod must lock both databases at the same time keep both data consistent and ensure that write operations, even with replication, are "all-or-nothing" operations.

9.4.10 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.

9.4.11 What kind of concurrency does MongoDB provide for JavaScript operations?

A single `mongod` can only run a *single* JavaScript operation at once. Therefore, operations that rely on JavaScript cannot run concurrently; however, the `mongod` can often run other database operations concurrently with the JavaScript execution. This limitation with JavaScript affects the following operations:

- `mapReduce`

  The JavaScript operations within a `mapReduce` job are short lived and yield many times during the operation. Portions of the map-reduce operation take database locks for reading, writing data to a temporary collection and writing the final output of the write operation.

- `group`

  The `group` takes a read lock in addition to blocking all other JavaScript execution.

- `db.eval()`

  Unless you specify the `nolock` option, `db.eval()` takes a write lock in addition to blocking all JavaScript operations.

- `$where`

  Only a single query that uses the `$where` operation can run at a time.

9.5 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 355) or post your question to the MongoDB User Mailing List[15].

[15] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

Frequently Asked Questions:

9.5.1 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.

9.5.2 How does sharding work with replication?

To use replication with sharding, deploy each *shard* as a *replica set*.

9.5.3 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 important of choosing a good *shard key* (page 296). 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 321) the shard key range to ensure initial even distribution.
- restore the dumped data into MongoDB.

See `shardCollection` (page 352), `sh.shardCollection()` (page 348), *Sharded Cluster Administration* (page 298), the *Shard Key* (page 304) section in the *Sharded Cluster Internals and Behaviors* (page 304) document, *Deploy a Sharded Cluster* (page 312), and SERVER-4000[16] for more information.

9.5.4 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.

9.5.5 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.

9.5.6 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.

9.5.7 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.

9.5.8 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.

[16] https://jira.mongodb.org/browse/SERVER-4000

- 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.

9.5.9 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.

9.5.10 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 54) works well in this case.

9.5.11 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.

9.5.12 Is it safe to remove old files in the `moveChunk` directory?

Yes. `mongod` creates these files as backups during normal *shard* balancing operations.

Once these migrations are complete, you may delete these files.

You can set `noMoveParanoia` to `true` to disable this behavior.

9.5.13 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 122) section of the *UNIX ulimit Settings* (page 122) document.

9.5.14 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.

9.5.15 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");
```

9.5.16 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.

9.5.17 How should administrators deal with failed migrations?

Failed migrations require no administrative intervention. Chunk moves are consistent and deterministic.

If a migration fails to complete for some reason, the *cluster* will retry the operation. When the migration completes successfully, the data will reside only on the new shard.

9.5.18 What is the process for moving, renaming, or changing the number of config servers?

See

Manage the Config Servers (page 318) which describes this process.

9.5.19 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.

9.5.20 Is it possible to quickly update mongos servers after updating a replica set configuration?

The mongos instances will detect these changes without intervention over time. However, if you want to force the mongos to reload its configuration, run the flushRouterConfig command against to each mongos directly.

9.5.21 What does the maxConns setting on mongos do?

The maxConns option limits the number of connections accepted by mongos.

If your client driver or application creates a large number of connections but allows them to time out rather than closing them explicitly, then it might make sense to limit the number of connections at the mongos layer.

Set maxConns to a value slightly higher than the maximum number of connections that the client creates, or the maximum size of the connection pool. This setting prevents the mongos from causing connection spikes on the individual *shards*. Spikes like these may disrupt the operation and memory allocation of the *sharded cluster*.

9.5.22 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[17] for more information.

9.5.23 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.

9.5.24 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.

9.5.25 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

[17] http://www.slideshare.net/mongodb/how-queries-work-with-sharding

of inserts. If most of your write operations are updates or read operations rather than inserts, this limitation should not impact your performance. However, if you have a high insert volume, this may be a limitation.

9.5.26 Can shard key be in ascending order, like dates or timestamps?

If you insert documents with monotonically increasing shard keys, all inserts will initially belong to the same *chunk* on a single *shard*. Although the system will eventually divide this chunk and migrate its contents to distribute data more evenly, at any moment the cluster can only direct insert operations at a single shard. This can limit the throughput of inserts.

If most of your write operations are updates or read operations rather than inserts, this limitation should not impact your performance. However, if you have a high insert volume, a monotonically increasing shard key may be a limitation.

To address this issue, you can use a field with a value that stores the hash of a key with an ascending value. While you can compute a hashed value in your application and include this value in your documents for use as a shard key, the SERVER-2001[18] issue will implement this capability within MongoDB.

9.5.27 What do `moveChunk commit failed` errors mean?

Consider the following error message:

```
ERROR: moveChunk commit failed: version is at <n>|<nn> instead of <N>|<NN>" and "ERROR: TERMINATING"
```

`mongod` issues this message if, during a *chunk migration* (page 309), the *shard* could not connect to the *config database* to update chunk information at the end of the migration process. If the shard cannot update the config database after `moveChunk`, the cluster will have an inconsistent view of all chunks. In these situations, the *primary* member of the shard will terminate itself to prevent data inconsistency. If the *secondary* member can access the config database, the shard's data will be accessible after an election. Administrators will need to resolve the chunk migration failure independently.

If you encounter this issue, contact the MongoDB User Group[19] or MongoDB support to address this issue.

9.5.28 How does draining a shard affect the balancing of uneven chunk distribution?

The sharded cluster balancing process controls both migrating chunks from decommissioned shards (i.e. draining,) and normal cluster balancing activities. Consider the following behaviors for different versions of MongoDB in situations where you remove a shard in a cluster with an uneven chunk distribution:

- After MongoDB 2.2, the balancer first removes the chunks from the draining shard and then balances the remaining uneven chunk distribution.
- Before MongoDB 2.2, the balancer handles the uneven chunk distribution and *then* removes the chunks from the draining shard.

9.6 FAQ: Replica Sets and Replication in MongoDB

This document answers common questions about database replication in MongoDB.

[18] https://jira.mongodb.org/browse/SERVER-2001
[19] http://groups.google.com/group/mongodb-user

If you don't find the answer you're looking for, check the *complete list of FAQs* (page 355) or post your question to the MongoDB User Mailing List[20].

Frequently Asked Questions:

9.6.1 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.

9.6.2 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.

9.6.3 What do the terms "secondary" and "slave" mean?

Secondary and *slave* nodes are read-only nodes that replicate from the *primary*.

Replication operates by way of an *oplog*, from which secondary/slave members apply new operations to themselves. This replication process is asynchronous, so secondary/slave nodes may not always reflect the latest writes to the primary. But usually, the gap between the primary and secondary nodes is just few milliseconds on a local network connection.

9.6.4 How long does replica set failover take?

It varies, but a replica set will select a new primary within a minute.

[20] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

It may take 10-30 seconds for the members of a *replica set* to declare a *primary* inaccessible. This triggers an *election*. During the election, the cluster is unavailable for writes.

The election itself may take another 10-30 seconds.

Note: *Eventually consistent* reads, like the ones that will return from a replica set are only possible with a *write concern* that permits reads from *secondary* members.

9.6.5 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 Distributed Replica Set (page 266)

9.6.6 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.

9.6.7 What is the preferred replication method: master/slave or replica sets?

New in version 1.8.

Replica sets are the preferred *replication* mechanism in MongoDB. However, if your deployment requires more than 12 nodes, you must use master/slave replication.

9.6.8 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.

9.6.9 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.

9.6.10 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 38). The `getLastError` command provides the facility for write concern. However, after the *default write concern change* (page 419), 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.

9.6.11 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 238).

If you have a three node replica set, you don't need an arbiter.

But a common configuration consists of two replicating nodes, one of which is *primary* and the other is *secondary*, as well as an arbiter for the third node. This configuration makes it possible for the set to elect a primary in the event of a failure without requiring three replicating nodes.

You may also consider adding an arbiter to a set if it has an equal number of nodes in two facilities and network partitions between the facilities are possible. In these cases, the arbiter will break the tie between the two facilities and allow the set to elect a new primary.

See also:

Replica Set Architectures and Deployment Patterns (page 238)

9.6.12 What information do arbiters exchange with the rest of the replica set?

Arbiters never receive the contents of a collection but do exchange the following data with the rest of the replica set:

- Credentials used to authenticate the arbiter with the replica set. All MongoDB processes within a replica set use keyfiles. These exchanges are encrypted.
- Replica set configuration data and voting data. This information is not encrypted. Only credential exchanges are encrypted.

If your MongoDB deployment uses SSL, then all communications between arbiters and the other members of the replica set are secure. See the documentation for *Use MongoDB with SSL Connections* (page 103) for more information. Run all arbiters on secure networks, as with all MongoDB components.

See

The overview of *Arbiter Members of Replica Sets* (page 226).

9.6.13 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 225), *hidden* (page 224) and *secondary-only* (page 224) members, as well as the *arbiters* (page 226).

See also:

Elections (page 218)

9.6.14 Do hidden members vote in replica set elections?

Hidden members (page 224) 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:

Elections (page 218)

9.6.15 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.

9.7 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 355) or post your question to the MongoDB User Mailing List[21].

Frequently Asked Questions:

- What are memory mapped files? (page 384)
- How do memory mapped files work? (page 384)
- How does MongoDB work with memory mapped files? (page 384)
- What are page faults? (page 384)
- What is the difference between soft and hard page faults? (page 384)
- What tools can I use to investigate storage use in MongoDB? (page 384)
- What is the working set? (page 384)
- Why are the files in my data directory larger than the data in my database? (page 385)
- How can I check the size of a collection? (page 386)
- How can I check the size of indexes? (page 386)
- How do I know when the server runs out of disk space? (page 387)

[21] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

9.7.1 What are memory mapped files?

A memory-mapped file is a file with data that the operating system places in memory by way of the `mmap()` system call. `mmap()` thus *maps* the file to a region of virtual memory. Memory-mapped files are the critical piece of the storage engine in MongoDB. By using memory mapped files MongoDB can treat the contents of its data files as if they were in memory. This provides MongoDB with an extremely fast and simple method for accessing and manipulating data.

9.7.2 How do memory mapped files work?

Memory mapping assigns files to a block of virtual memory with a direct byte-for-byte correlation. Once mapped, the relationship between file and memory allows MongoDB to interact with the data in the file as if it were memory.

9.7.3 How does MongoDB work with memory mapped files?

MongoDB uses memory mapped files for managing and interacting with all data. MongoDB memory maps data files to memory as it accesses documents. Data that isn't accessed is *not* mapped to memory.

9.7.4 What are page faults?

Page faults will occur if you're attempting to access part of a memory-mapped file that *isn't* in memory.

If there is free memory, then the operating system can find the page on disk and load it to memory directly. However, if there is no free memory, the operating system must:

- find a page in memory that is stale or no longer needed, and write the page to disk.
- read the requested page from disk and load it into memory.

This process, particularly on an active system can take a long time, particularly in comparison to reading a page that is already in memory.

9.7.5 What is the difference between soft and hard page faults?

Page faults occur when MongoDB needs access to data that isn't currently in active memory. A "hard" page fault refers to situations when MongoDB must access a disk to access the data. A "soft" page fault, by contrast, merely moves memory pages from one list to another, such as from an operating system file cache. In production, MongoDB will rarely encounter soft page faults.

9.7.6 What tools can I use to investigate storage use in MongoDB?

The `db.stats()` method in the `mongo` shell, returns the current state of the "active" database. The `http://docs.mongodb.org/manual/reference/database-statistics` document describes the fields in the `db.stats()` output.

9.7.7 What is the working set?

Working set represents the total body of data that the application uses in the course of normal operation. Often this is a subset of the total data size, but the specific size of the working set depends on actual moment-to-moment use of the database.

If you run a query that requires MongoDB to scan every document in a collection, the working set will expand to include every document. Depending on physical memory size, this may cause documents in the working set to "page out," or to be removed from physical memory by the operating system. The next time MongoDB needs to access these documents, MongoDB may incur a hard page fault.

If you run a query that requires MongoDB to scan every *document* in a collection, the working set includes every active document in memory.

For best performance, the majority of your *active* set should fit in RAM.

9.7.8 Why are the files in my data directory larger than the data in my database?

The data files in your data directory, which is the `/data/db` directory in default configurations, might be larger than the data set inserted into the database. Consider the following possible causes:

- Preallocated data files.

  In the data directory, MongoDB preallocates data files to a particular size, in part to prevent file system fragmentation. MongoDB names the first data file `<databasename>.0`, the next `<databasename>.1`, etc. The first file `mongod` allocates is 64 megabytes, the next 128 megabytes, and so on, up to 2 gigabytes, at which point all subsequent files are 2 gigabytes. The data files include files with allocated space but that hold no data. `mongod` may allocate a 1 gigabyte data file that may be 90% empty. For most larger databases, unused allocated space is small compared to the database.

  On Unix-like systems, `mongod` preallocates an additional data file and initializes the disk space to `0`. Preallocating data files in the background prevents significant delays when a new database file is next allocated.

  You can disable preallocation with the `noprealloc` run time option. However `noprealloc` is **not** intended for use in production environments: only use `noprealloc` for testing and with small data sets where you frequently drop databases.

  On Linux systems you can use `hdparm` to get an idea of how costly allocation might be:

  ```
  time hdparm --fallocate $((1024*1024)) testfile
  ```

- The *oplog*.

  If this `mongod` is a member of a replica set, the data directory includes the *oplog.rs* file, which is a preallocated *capped collection* in the `local` database. The default allocation is approximately 5% of disk space on 64-bit installations, see *Oplog Sizing* (page 220) for more information. In most cases, you should not need to resize the oplog. However, if you do, see *Change the Size of the Oplog* (page 271).

- The *journal*.

  The data directory contains the journal files, which store write operations on disk prior to MongoDB applying them to databases. See *Journaling* (page 100).

- Empty records.

  MongoDB maintains lists of empty records in data files when deleting documents and collections. MongoDB can reuse this space, but will never return this space to the operating system.

  To de-fragment allocated storage, use `compact`, which de-fragments allocated space. By de-fragmenting storage, MongoDB can effectively use the allocated space. `compact` requires up to 2 gigabytes of extra disk space to run. Do not use `compact` if you are critically low on disk space.

  Important: `compact` only removes fragmentation from MongoDB data files and does not return any disk space to the operating system.

To reclaim deleted space, use `repairDatabase`, which rebuilds the database which de-fragments the storage and may release space to the operating systme. `repairDatabase` requires up to 2 gigabytes of extra disk space to run. Do not use `repairDatabase` if you are critically low on disk space.

Warning: `repairDatabase` requires enough free disk space to hold both the old and new database files while the repair is running. Be aware that `repairDatabase` will block all other operations and may take a long time to complete.

9.7.9 How can I check the size of a collection?

To view the size of a collection and other information, use the `stats()` method from the `mongo` shell. The following example issues `stats()` for the `orders` collection:

```
db.orders.stats();
```

To view specific measures of size, use these methods:

- `db.collection.dataSize()`: data size for the collection.
- `db.collection.storageSize()`: allocation size, including unused space.
- `db.collection.totalSize()`: the data size plus the index size.
- `db.collection.totalIndexSize()`: the index size.

Also, the following scripts print the statistics for each database and collection:

```
db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); prin

db._adminCommand("listDatabases").databases.forEach(function (d) {mdb = db.getSiblingDB(d.name); mdb
```

9.7.10 How can I check the size of indexes?

To view the size of the data allocated for an index, use one of the following procedures in the `mongo` shell:

- Use the `stats()` method using the index namespace. To retrieve a list of namespaces, issue the following command:

  ```
  db.system.namespaces.find()
  ```

- Check the value of `indexSizes` in the output of the `db.collection.stats()` command.

Example

Issue the following command to retrieve index namespaces:

```
db.system.namespaces.find()
```

The command returns a list similar to the following:

```
{"name" : "test.orders"}
{"name" : "test.system.indexes"}
{"name" : "test.orders.$_id_"}
```

View the size of the data allocated for the `orders.$_id_` index with the following sequence of operations:

```
use test
db.orders.$_id_.stats().indexSizes
```

9.7.11 How do I know when the server runs out of disk space?

If your server runs out of disk space for data files, you will see something like this in the log:

```
Thu Aug 11 13:06:09 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:09 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:09 [FileAllocator]     will try again in 10 seconds
Thu Aug 11 13:06:19 [FileAllocator] allocating new data file dbms/test.13, filling with zeroes...
Thu Aug 11 13:06:19 [FileAllocator] error failed to allocate new file: dbms/test.13 size: 2146435072
Thu Aug 11 13:06:19 [FileAllocator]     will try again in 10 seconds
```

The server remains in this state forever, blocking all writes including deletes. However, reads still work. To delete some data and compact, using the `compact` command, you must restart the server first.

If your server runs out of disk space for journal files, the server process will exit. By default, `mongod` creates journal files in a sub-directory of `dbpath` named `journal`. You may elect to put the journal files on another storage device using a filesystem mount or a symlink.

Note: If you place the journal files on a separate storage device you will not be able to use a file system snapshot tool to capture a consistent snapshot of your data files and journal files.

9.8 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 355) or post your question to the MongoDB User Mailing List[22]. See also *Indexing Strategies* (page 199).

Frequently Asked Questions:

9.8.1 Should you run `ensureIndex()` after every insert?

No. You only need to create an index once for a single collection. After initial creation, MongoDB automatically updates the index as data changes.

While running `ensureIndex()` is usually ok, if an index doesn't exist because of ongoing administrative work, a call to `ensureIndex()` may disrupt database availability. Running `ensureIndex()` can render a replica set inaccessible as the index creation is happening. See *Build Indexes on Replica Sets* (page 198).

[22] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

9.8.2 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[23].

9.8.3 How do you determine the size of an index?

To check the sizes of the indexes on a collection, use `db.collection.stats()`.

9.8.4 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 203).

9.8.5 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.

9.8.6 How do you determine what fields to index?

A number of factors determine what fields to index, including *selectivity* (page 203), fitting indexes into RAM, reusing indexes in multiple queries when possible, and creating indexes that can support all the fields in a given query. For detailed documentation on choosing which fields to index, see *Indexing Strategies* (page 199).

9.8.7 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.

9.8.8 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 Options* (page 191).

If you build a large index without the background option, and if doing so causes the database to stop responding, wait for the index to finish building.

[23] http://api.mongodb.org/

9.8.9 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.

9.8.10 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 203). If you need to use these, it is often best to make sure that an additional, more selective criterion is part of the query.

9.8.11 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.

9.8.12 How can I effectively use indexes strategy for attribute lookups?

For simple attribute lookups that don't require sorted result sets or range queries, consider creating a field that contains an array of documents where each document has a field (e.g. `attrib`) that holds a specific type of attribute. You can index this `attrib` field.

For example, the `attrib` field in the following document allows you to add an unlimited number of attributes types:

```
{ _id : ObjectId(...),
  attrib : [
             { k: "color", v: "red" },
             { k: "shape": v: "rectangle" },
             { k: "color": v: "blue" },
             { k: "avail": v: true }
           ]
}
```

Both of the following queries could use the same `{ "attrib.k": 1, "attrib.v": 1 }` index:

```
db.mycollection.find( { attrib: { $elemMatch : { k: "color", v: "blue" } } } )
db.mycollection.find( { attrib: { $elemMatch : { k: "avail", v: true } } } )
```

9.9 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 355) or post your question to the MongoDB User Mailing List[24].

[24] https://groups.google.com/forum/?fromgroups#!forum/mongodb-user

Frequently Asked Questions:

9.9.1 Where can I find information about a mongod process that stopped running unexpectedly?

If mongod shuts down unexpectedly on a UNIX or UNIX-based platform, and if mongod fails to log a shutdown or error message, then check your system logs for messages pertaining to MongoDB. For example, for logs located in /var/log/messages, use the following commands:

```
sudo grep mongod /var/log/messages
sudo grep score /var/log/messages
```

9.9.2 Does TCP keepalive time affect sharded clusters and replica sets?

If you experience socket errors between members of a sharded cluster or replica set, that do not have other reasonable causes, check the TCP keep alive value, which Linux systems store as the tcp_keepalive_time value. A common keep alive period is 7200 seconds (2 hours); however, different distributions and OS X may have different settings. For MongoDB, you will have better experiences with shorter keepalive periods, on the order of 300 seconds (five minutes).

On Linux systems you can use the following operation to check the value of tcp_keepalive_time:

```
cat /proc/sys/net/ipv4/tcp_keepalive_time
```

You can change the tcp_keepalive_time value with the following operation:

```
echo 300 > /proc/sys/net/ipv4/tcp_keepalive_time
```

The new tcp_keepalive_time value takes effect without requiring you to restart the mongod or mongos servers. When you reboot or restart your system you will need to set the new tcp_keepalive_time value, or see your operating system's documentation for setting the TCP keepalive value persistently.

For OS X systems, issue the following command to view the keep alive setting:

```
sysctl net.inet.tcp.keepinit
```

To set a shorter keep alive period use the following invocation:

```
sysctl -w net.inet.tcp.keepinit=300
```

If your replica set or sharded cluster experiences keepalive-related issues, you must alter the `tcp_keepalive_time` value on all machines hosting MongoDB processes. This includes all machines hosting `mongos` or `mongod` servers.

Windows users should consider the Windows Server Technet Article on KeepAliveTime configuration[25] for more information on setting keep alive for MongoDB deployments on Windows systems.

9.9.3 What tools are available for monitoring MongoDB?

The *MongoDB Management Services <http://mms.mongodb.com>* includes monitoring. MMS Monitoring is a free, hosted services for monitoring MongoDB deployments. A full list of third-party tools is available as part of the *Monitoring Database Systems* (page 109) documentation. Also consider the MMS Documentation[26].

9.9.4 Memory Diagnostics

Do I need to configure swap space?

Always configure systems to have swap space. Without swap, your system may not be reliant in some situations with extreme memory constraints, memory leaks, or multiple programs using the same memory. Think of the swap space as something like a steam release valve that allows the system to release extra pressure without affecting the overall functioning of the system.

Nevertheless, systems running MongoDB *do not* need swap for routine operation. Database files are *memory-mapped* (page 384) 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.

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 383) and working set.
- The operating system's cache strategy for LRU (Least Recently Used)
- The impact of *journaling* (page 100)
- The number or rate of page faults and other MMS gauges to detect when you need more RAM

[25] http://technet.microsoft.com/en-us/library/dd349797.aspx#BKMK_2
[26] http://mms.mongodb.com/help/

MongoDB defers to the operating system when loading data into memory from disk. It simply *memory maps* (page 384) all its data files and relies on the operating system to cache data. The OS typically evicts the least-recently-used data from RAM when it runs low on memory. For example if clients access indexes more frequently than documents, then indexes will more likely stay in RAM, but it depends on your particular usage.

To calculate how much RAM you need, you must calculate your working set size, or the portion of your data that clients use most often. This depends on your access patterns, what indexes you have, and the size of your documents.

If page faults are infrequent, your working set fits in RAM. If fault rates rise higher than that, you risk performance degradation. This is less critical with SSD drives than with spinning disks.

How do I read memory statistics in the UNIX `top` command

Because `mongod` uses *memory-mapped files* (page 384), 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`.

9.9.5 Sharded Cluster Diagnostics

The two most important factors in maintaining a successful sharded cluster are:

- *choosing an appropriate shard key* (page 304) and
- *sufficient capacity to support current and future operations* (page 297).

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 308). 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 309) 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 320).

Finally, if your shard key has a low *cardinality* (page 305), MongoDB may not be able to create sufficient splits among the data.

Why would one shard receive a disproportion amount of traffic in a sharded cluster?

In some situations, a single shard or a subset of the cluster will receive a disproportionate portion of the traffic and workload. In almost all cases this is the result of a shard key that does not effectively allow *write scaling* (page 305).

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 307) 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 392).

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 305) 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 327) that is too short, given the rate of data growth.
  - an uneven distribution of *write operations* (page 305) 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 327) 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 322) 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 315) 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 305).

CHAPTER 10

Release Notes

Always install the latest, stable version of MongoDB. See *Version Numbers* (page 420) for more information.

See the following release notes for an account of the changes in major versions. Release notes also include instructions for upgrade.

10.1 Current Stable Release

(*2.2-series*)

10.1.1 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.

Always upgrade to the latest point release in the 2.2 point release. Currently the latest release of MongoDB is 2.2.7.

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 328) 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[1].
2. Shutdown your `mongod` instance. Replace the existing binary with the 2.2 `mongod` binary and restart MongoDB.

Upgrading a Replica Set

You can upgrade to 2.2 by performing a "rolling" upgrade of the set by upgrading the members individually while the other members are available to minimize downtime. Use the following procedure:

1. Upgrade the *secondary* members of the set one at a time by shutting down the `mongod` and replacing the 2.0 binary with the 2.2 binary. After upgrading a `mongod` instance, wait for the member to recover to `SECONDARY` state before upgrading the next instance. To check the member's state, issue `rs.status()` (page 286) in the `mongo` shell.
2. Use the `mongo` shell method `rs.stepDown()` (page 288) to step down the *primary* to allow the normal *failover* (page 236) procedure. `rs.stepDown()` (page 288) expedites the failover procedure and is preferable to shutting down the primary directly.

   Once the primary has stepped down and another member has assumed `PRIMARY` state, as observed in the output of `rs.status()` (page 286), 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 328).
- Upgrade all `mongos` instances *first*, in any order.
- Upgrade all of the `mongod` config server instances using the *stand alone* (page 396) procedure. To keep the cluster online, be sure that at all times at least one config server is up.
- Upgrade each shard's replica set, using the *upgrade procedure for replica sets* (page 396) 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[2] for more information.

---

[1] http://downloads.mongodb.org/
[2] https://jira.mongodb.org/browse/SERVER-6902

Changes

Major Features

Aggregation Framework The aggregation framework makes it possible to do aggregation operations without needing to use *map-reduce*. The `aggregate` command exposes the aggregation framework, and the `db.collection.aggregate()` helper in the `mongo` shell provides an interface to these operations. Consider the following resources for background on the aggregation framework and its use:

- Documentation: *Aggregation Framework* (page 151)
- Reference: *Aggregation Framework Reference* (page 163)
- Examples: *Aggregation Framework Examples* (page 155)

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 `http://docs.mongodb.org/manual/tutorial/expire-data` tutorial.

Concurrency Improvements MongoDB 2.2 increases the server's capacity for concurrent operations with the following improvements:

1. DB Level Locking[3]
2. Improved Yielding on Page Faults[4]
3. Improved Page Fault Detection on Windows[5]

To reflect these changes, MongoDB now provides changed and improved reporting for concurrency and use, see *locks* and *server-status-record-stats* in `server status` and see `current operation output`, `db.currentOp()`, `mongotop`, and `mongostat`.

Improved Data Center Awareness with Tag Aware Sharding MongoDB 2.2 adds additional support for geographic distribution or other custom partitioning for sharded collections in *clusters*. By using this "tag aware" sharding, you can automatically ensure that data in a sharded database system is always on specific shards. For example, with tag aware sharding, you can ensure that data is closest to the application servers that use that data most frequently.

Shard tagging controls data location, and is complementary but separate from replica set tagging, which controls *read preference* (page 243) and *write concern* (page 38). 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()` (page 350)
- `sh.addTagRange()` (page 350)
- `sh.removeShardTag()` (page 351)

Also, see *Tag Aware Sharding* (page 336).

[3] https://jira.mongodb.org/browse/SERVER-4328
[4] https://jira.mongodb.org/browse/SERVER-3357
[5] https://jira.mongodb.org/browse/SERVER-4538

Fully Supported Read Preference Semantics All MongoDB clients and drivers now support full *read preferences* (page 243), including consistent support for a full range of *read preference modes* (page 244) and *tag sets* (page 246). This support extends to the `mongos` and applies identically to single replica sets and to the replica sets for each shard in a *sharded cluster*.

Additional read preference support now exists in the `mongo` shell using the `readPref()` cursor method.

Compatibility Changes

Authentication Changes MongoDB 2.2 provides more reliable and robust support for authentication clients, including drivers and `mongos` instances.

If your cluster runs with authentication:

- For all drivers, use the latest release of your driver and check its release notes.
- In sharded environments, to ensure that your cluster remains available during the upgrade process you **must** use the *upgrade procedure for sharded clusters* (page 396).

`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[6] 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[7] for more information.

`ObjectId().toString()` Returns String Literal `ObjectId("...")` In version 2.2, the `ObjectId.toString()` method returns the string representation of the *ObjectId()* (page 55) object and has the format `ObjectId("...")`.

Consider the following example that calls the `toString()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").toString()
```

The method now returns the *string* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

Previously, in version 2.0, the method would return the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str* (page 55), which holds the hexadecimal string value in both versions.

[6] https://jira.mongodb.org/browse/SERVER-6226
[7] https://jira.mongodb.org/browse/SERVER-6961

`ObjectId().valueOf()` Returns hexadecimal string In version 2.2, the `ObjectId.valueOf()` method returns the value of the *ObjectId()* (page 55) object as a lowercase hexadecimal string.

Consider the following example that calls the `valueOf()` method on the `ObjectId("507c7f79bcf86cd7994f6c0e")` object:

```
ObjectId("507c7f79bcf86cd7994f6c0e").valueOf()
```

The method now returns the *hexadecimal string* `507c7f79bcf86cd7994f6c0e`.

Previously, in version 2.0, the method would return the *object* `ObjectId("507c7f79bcf86cd7994f6c0e")`.

If compatibility between versions 2.0 and 2.2 is required, use *ObjectId().str* (page 55) attribute, which holds the hexadecimal string value in both versions.

Behavioral Changes

Restrictions on Collection Names In version 2.2, collection names cannot:

- contain the `$`.
- be an empty string (i.e. `""`).

This change does not affect collections created with now illegal names in earlier versions of MongoDB.

These new restrictions are in addition to the existing restrictions on collection names which are:

- A collection name should begin with a letter or an underscore.
- A collection name cannot contain the null character.
- Begin with the `system.` prefix. MongoDB reserves `system.` for system collections, such as the `system.indexes` collection.
- The maximum size of a collection name is 128 characters, including the name of the database. However, for maximum flexibility, collections should have names less than 80 characters.

Collections names may have any other valid UTF-8 string.

See the SERVER-4442[8] and the *Are there any restrictions on the names of Collections?* (page 366) FAQ item.

Restrictions on Database Names for Windows Database names running on Windows can no longer contain the following characters:

```
/\. "*<>:|?
```

The names of the data files include the database name. If you attempt to upgrade a database instance with one or more of these characters, `mongod` will refuse to start.

Change the name of these databases before upgrading. See SERVER-4584[9] and SERVER-6729[10] 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[11] for more information.

[8] https://jira.mongodb.org/browse/SERVER-4442
[9] https://jira.mongodb.org/browse/SERVER-4584
[10] https://jira.mongodb.org/browse/SERVER-6729
[11] https://jira.mongodb.org/browse/SERVER-5516

New `$elemMatch` Projection Operator The `$elemMatch` operator allows applications to narrow the data returned from queries so that the query operation will only return the first matching element in an array. See the `http://docs.mongodb.org/manual/reference/operator/projection/elemMatch` documentation and the SERVER-2238[12] and SERVER-828[13] 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[14] for more information.

Service Support for `mongos.exe` You may now run `mongos.exe` instances as a Windows Service. See the `http://docs.mongodb.org/manual/reference/mongos.exe` reference and *MongoDB as a Windows Service* (page 16) and SERVER-1589[15] for more information.

Log Rotate Command Support MongoDB for Windows now supports log rotation by way of the `logRotate` database command. See SERVER-2612[16] for more information.

New Build Using SlimReadWrite Locks for Windows Concurrency Labeled "2008+" on the Downloads Page[17], this build for 64-bit versions of Windows Server 2008 R2 and for Windows 7 or newer, offers increased performance over the standard 64-bit Windows build of MongoDB. See SERVER-3844[18] 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-808[19] for more information.

`mongorestore` now includes the *`--noIndexRestore`* option to provide the preceding behavior. Use *`--noIndexRestore`* to prevent `mongorestore` from building previous indexes.

`mongooplog` for Replaying Oplogs The `mongooplog` tool makes it possible to pull *oplog* entries from `mongod` instance and apply them to another `mongod` instance. You can use `mongooplog` to achieve point-in-time backup of a MongoDB data set. See the SERVER-3873[20] case and the `http://docs.mongodb.org/manual/reference/mongooplog` documentation.

Authentication Support for `mongotop` and `mongostat` `mongotop` and `mongostat` now contain support for username/password authentication. See SERVER-3875[21] and SERVER-3871[22] for more information regarding this change. Also consider the documentation of the following options for additional information:

[12] https://jira.mongodb.org/browse/SERVER-2238
[13] https://jira.mongodb.org/browse/SERVER-828
[14] https://jira.mongodb.org/browse/SERVER-5648
[15] https://jira.mongodb.org/browse/SERVER-1589
[16] https://jira.mongodb.org/browse/SERVER-2612
[17] http://www.mongodb.org/downloads
[18] https://jira.mongodb.org/browse/SERVER-3844
[19] https://jira.mongodb.org/browse/SERVER-808
[20] https://jira.mongodb.org/browse/SERVER-3873
[21] https://jira.mongodb.org/browse/SERVER-3875
[22] https://jira.mongodb.org/browse/SERVER-3871

- *mongotop --username*
- *mongotop --password*
- *mongostat --username*
- *mongostat --password*

Write Concern Support for `mongoimport` and `mongorestore` `mongoimport` now provides an option to halt the import if the operation encounters an error, such as a network interruption, a duplicate key exception, or a write error. The *--stopOnError* option will produce an error rather than silently continue importing data. See SERVER-3937[23] for more information.

In `mongorestore`, the *--w* option provides support for configurable write concern.

`mongodump` Support for Reading from Secondaries You can now run `mongodump` when connected to a *secondary* member of a *replica set*. See SERVER-3854[24] for more information.

`mongoimport` Support for full 16MB Documents Previously, `mongoimport` would only import documents that were less than 4 megabytes in size. This issue is now corrected, and you may use `mongoimport` to import documents that are at least 16 megabytes ins size. See SERVER-4593[25] for more information.

`Timestamp()` Extended JSON format MongoDB extended JSON now includes a new `Timestamp()` type to represent the Timestamp type that MongoDB uses for timestamps in the *oplog* among other contexts.

This permits tools like `mongooplog` and `mongodump` to query for specific timestamps. Consider the following `mongodump` operation:

```
mongodump --db local --collection oplog.rs --query '{"ts":{"$gt":{"$timestamp" : {"t": 1344969612000,
```

See SERVER-3483[26] 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[27] for more information.
- Multi-line command support in shell history. See SERVER-3470[28] for more information.
- Windows support for the `edit` command. See SERVER-3998[29] 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[30] for more information.

[23] https://jira.mongodb.org/browse/SERVER-3937
[24] https://jira.mongodb.org/browse/SERVER-3854
[25] https://jira.mongodb.org/browse/SERVER-4593
[26] https://jira.mongodb.org/browse/SERVER-3483
[27] https://jira.mongodb.org/browse/SERVER-4312
[28] https://jira.mongodb.org/browse/SERVER-3470
[29] https://jira.mongodb.org/browse/SERVER-3998
[30] https://jira.mongodb.org/browse/SERVER-1651

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[31] and SERVER-2395[32] 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[33] case and the documentation of the `syslog` run-time option or the *`mongod --syslog`* and *`mongos --syslog`* command line-options.

`touch` Command Added the `touch` command to read the data and/or indexes from a collection into memory. See: SERVER-2023[34] 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[35] and `indexCounters` for more information.

Padding Specifiable on `compact` Command See the documentation of the `compact` and the SERVER-4018[36] 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[37] and SERVER-5172[38] 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[39] and SERVER-4683[40]. For more information about TCMalloc, see the documentation of TCMalloc[41] itself.

[31] https://jira.mongodb.org/browse/SERVER-3819
[32] https://jira.mongodb.org/browse/SERVER-2395
[33] https://jira.mongodb.org/browse/SERVER-2957
[34] https://jira.mongodb.org/browse/SERVER-2023
[35] https://jira.mongodb.org/browse/SERVER-5784
[36] https://jira.mongodb.org/browse/SERVER-4018
[37] https://jira.mongodb.org/browse/SERVER-3829
[38] https://jira.mongodb.org/browse/SERVER-5172
[39] https://jira.mongodb.org/browse/SERVER-188
[40] https://jira.mongodb.org/browse/SERVER-4683
[41] http://goog-perftools.sourceforge.net/doc/tcmalloc.html

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[42] for more information.

Replica Set Members can Sync from Specific Members The new `replSetSyncFrom` (page 293) command and new `rs.syncFrom()` (page 289) 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` (page 293) when overriding the default behavior.

Replica Set Members will not Sync from Members Without Indexes Unless `buildIndexes: false` To prevent inconsistency between members of replica sets, if the member of a replica set has `buildIndexes` 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[43] for more information.

New Option To Configure Index Pre-Fetching during Replication By default, when replicating options, *secondaries* will pre-fetch *Indexes* (page 185) associated with a query to improve replication throughput in most cases. The `replIndexPrefetch` setting and `--replIndexPrefetch` option allow administrators to disable this feature or allow the `mongod` to pre-fetch only the index on the `_id` field. See SERVER-6718[44] for more information.

Map Reduce Improvements

In 2.2 Map Reduce received the following improvements:

- Improved support for sharded MapReduce[45], and
- MapReduce will retry jobs following a config error[46].

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 307)" documentation and SERVER-1506[47] for more information.

Migration Thresholds Modified The *migration thresholds* (page 308) 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 308) documentation for more information.

[42] https://jira.mongodb.org/browse/SERVER-3575
[43] https://jira.mongodb.org/browse/SERVER-4160
[44] https://jira.mongodb.org/browse/SERVER-6718
[45] https://jira.mongodb.org/browse/SERVER-4521
[46] https://jira.mongodb.org/browse/SERVER-4158
[47] https://jira.mongodb.org/browse/SERVER-1506

Licensing Changes

Added License notice for Google Perftools (TCMalloc Utility). See the License Notice[48] and the SERVER-4683[49] for more information.

Resources

- MongoDB Downloads[50].
- All JIRA issues resolved in 2.2[51].
- All backwards incompatible changes[52].
- All third party license notices[53].
- What's New in MongoDB 2.2 Online Conference[54].

See `http://docs.mongodb.org/manual/release-notes/2.2-changes` for an overview of all changes in 2.2.

10.2 Previous Stable Releases

10.2.1 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[55].

:wiki:'Journaling' is **enabled by default** in 2.0 for 64-bit builds. If you still prefer to run without journaling, start `mongod` with the `--nojournal` 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:

[48] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES#L231
[49] https://jira.mongodb.org/browse/SERVER-4683
[50] http://mongodb.org/downloads
[51] https://jira.mongodb.org/secure/IssueNavigator.jspa?reset=true&jqlQuery=project+%3D+SERVER+AND+fixVersion+in+%28%222.1.0%22%2C+%222.1.1%22%2rc0%22%2C+%222.2.0-rc1%22%2C+%222.2.0-rc2%22%29+ORDER+BY+component+ASC%2C+key+DESC
[52] https://jira.mongodb.org/secure/IssueNavigator.jspa?requestId=11225
[53] https://github.com/mongodb/mongo/blob/v2.2/distsrc/THIRD-PARTY-NOTICES
[54] http://www.mongodb.com/events/webinar/mongodb-online-conference-sept
[55] https://jira.mongodb.org/browse/SERVER-1097

Upgrading a Standalone `mongod`

1. Download the v2.0.x binaries from the MongoDB Download Page[56].
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[57].
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 38) in your application code to confirm that each update reaches multiple servers.
3. Use the `rs.stepDown()` (page 288) to step down the primary to allow the normal *failover* (page 236) procedure.

   `rs.stepDown()` (page 288) and `replSetStepDown` provide for shorter and more consistent failover procedures than simply shutting down the primary directly.

   When the primary has stepped down, shut down its instance and upgrade by replacing the `mongod` binary with the 2.0.x binary.

Upgrading a Sharded Cluster

1. Upgrade all *config server* instances *first*, in any order. Since config servers use two-phase commit, *shard* configuration metadata updates will halt until all are up and running.
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[58] for more information.

The specific operations yield in 2.0 are:

- Updates by `_id`
- Removes

[56] http://downloads.mongodb.org/
[57] http://downloads.mongodb.org/
[58] https://jira.mongodb.org/browse/SERVER-2563

- 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`. 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[59].
- The `repairDatabase` command converts indexes to a 2.0 indexes.

To convert all indexes for a given collection to the *2.0 type* (page 406), 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 `http://docs.mongodb.org/manual/tutorial/roll-back-to-v1.8-index`.

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`.

[59] https://jira.mongodb.org/browse/SERVER-3866

During normal operation, the set will always chose `B` as primary. If `B` becomes unavailable, the set will elect `A` as primary.

For more information, see the *Member Priority* (page 219) 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 38) across data centers, racks, specific servers, or any other architecture choice.

For example, an administrator can define rules such as "very important write" or `customerData` or "audit-trail" to replicate to certain servers, racks, data centers, etc. Then in the application code, the developer would say:

```
db.foo.insert(doc, {w : "very important write"})
```

which would succeed if it fulfilled the conditions the DBA defined for "very important write".

For more information, see **:wiki:'Tagging <Data+Center+Awareness#DataCenterAwareness-Tagging%28version2.0%29>'**.

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 `http://docs.mongodb.org/manual/applications/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 241).

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 281).

Primary Checks for a Caught up Secondary before Stepping Down To minimize time without a *primary*, the `rs.stepDown()` (page 288) 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 273).

Extended Shutdown on the Primary to Minimize Interruption When you call the `shutdown` command, the *primary* will refuse to shut down unless there is a *secondary* whose optime is within 10 seconds of the primary. If such a secondary isn't available, the primary will step down and wait up to a minute for the secondary to be fully caught up before shutting down.

Note that to get this behavior, you must issue the `shutdown` command explicitly; sending a signal to the process will not trigger this behavior.

You can also force the primary to shut down, even without an up-to-date secondary available.

Maintenance Mode When `repair` or `compact` runs on a *secondary*, the secondary will automatically drop into "recovering" mode until the operation finishes. This prevents clients from trying to read from it while it's busy.

Geospatial Features

Multi-Location Documents Indexing is now supported on documents which have multiple location objects, embedded either inline or in nested sub-documents. Additional command options are also supported, allowing results to return with not only distance but the location used to generate the distance.

For more information, see **:wiki:'Multi-location Documents <Geospatial+Indexing#GeospatialIndexing-MultilocationDocuments>'**.

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.

See **:wiki:'OP_INSERT <Mongo+Wire+Protocol#MongoWireProtocol-OPINSERT>'**.

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 **:wiki:'MapReduce Output Options <MapReduce#MapReduce-Outputoptions>'** and `http://docs.mongodb.org/manual/reference/command/mapReduce`.

Performance Improvements Map/reduce performance will benefit from the following:

- Larger in-memory buffer sizes, reducing the amount of disk I/O needed during a job
- Larger javascript heap size, allowing for larger objects and less GC

- Supports pure JavaScript execution with the `jsMode` flag. See `http://docs.mongodb.org/manual/reference/command/mapReduce`.

New Querying Features

Additional regex options: `s` Allows the dot (`.`) to match all characters including new lines. This is in addition to the currently supported `i`, `m` and `x`. See **:wiki:'Regular Expressions <Advanced+Queries#AdvancedQueries-RegularExpressions>'** and `$regex`.

$and A special boolean `$and` query operator is now available.

Command Output Changes

The output of the `validate` command and the documents in the `system.profile` collection have both been enhanced to return information as BSON objects with keys for each value rather than as free-form strings.

Shell Features

Custom Prompt You can define a custom prompt for the `mongo` shell. You can change the prompt at any time by setting the prompt variable to a string or a custom JavaScript function returning a string. For examples, see **:wiki:'Custom Prompt <Overview+-+The+MongoDB+Interactive+Shell#Overview-TheMongoDBInteractiveShell-CustomPrompt>'**.

Default Shell Init Script On startup, the shell will check for a `.mongorc.js` file in the user's home directory. The shell will execute this file after connecting to the database and before displaying the prompt.

If you would like the shell not to run the `.mongorc.js` file automatically, start the shell with *`--norc`*.

For more information, see `http://docs.mongodb.org/manual/reference/mongo`.

Most Commands Require Authentication

In 2.0, when running with authentication (e.g. `auth`) *all* database commands require authentication, *except* the following commands.

- `isMaster` (page 289)
- `authenticate`
- `getnonce`
- `buildInfo`
- `ping`
- `isdbgrid`

Resources

- MongoDB Downloads[60]

[60] http://mongodb.org/downloads

- All JIRA Issues resolved in 2.0[61]
- All Backward Incompatible Changes[62]

10.2.2 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:

- *Replica set* members should be upgraded in a particular order, as described in *Upgrading a Replica Set* (page 410).
- 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[63].
2. Shutdown your `mongod` instance.
3. Replace the existing binary with the 1.8.x `mongod` binary.
4. Restart MongoDB.

Upgrading a Replica Set

1.8.x *secondaries* **can** replicate from 1.6.x *primaries*.

1.6.x secondaries **cannot** replicate from 1.8.x primaries.

Thus, to upgrade a *replica set* you must replace all of your secondaries first, then the primary.

For example, suppose you have a replica set with a primary, an *arbiter* and several secondaries. To upgrade the set, do the following:

1. For the arbiter:
   (a) Shut down the arbiter.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[64].

[61] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=11002
[62] https://jira.mongodb.org/secure/IssueNavigator.jspa?requestId=11023
[63] http://downloads.mongodb.org/
[64] http://downloads.mongodb.org/

2. Change your config (optional) to prevent election of a new primary.

   It is possible that, when you start shutting down members of the set, a new primary will be elected. To prevent this, you can give all of the secondaries a priority of `0` before upgrading, and then change them back afterwards. To do so:

   (a) Record your current config. Run `rs.config()` (page 286) and paste the results into a text file.

   (b) Update your config so that all secondaries have priority `0`. For example:

```
config = rs.conf()
{
      "_id" : "foo",
      "version" : 3,
      "members" : [
              {
                      "_id" : 0,
                      "host" : "ubuntu:27017"
              },
              {
                      "_id" : 1,
                      "host" : "ubuntu:27018"
              },
              {
                      "_id" : 2,
                      "host" : "ubuntu:27019",
                      "arbiterOnly" : true
              }
              {
                      "_id" : 3,
                      "host" : "ubuntu:27020"
              },
              {
                      "_id" : 4,
                      "host" : "ubuntu:27021"
              },
      ]
}
config.version++
3
rs.isMaster()
{
      "setName" : "foo",
      "ismaster" : false,
      "secondary" : true,
      "hosts" : [
              "ubuntu:27017",
              "ubuntu:27018"
      ],
      "arbiters" : [
              "ubuntu:27019"
      ],
      "primary" : "ubuntu:27018",
      "ok" : 1
}
// for each secondary
config.members[0].priority = 0
config.members[3].priority = 0
config.members[4].priority = 0
rs.reconfig(config)
```

3. For each secondary:
   (a) Shut down the secondary.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[65].
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[66].

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 410).
   - 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[67].
3. For each `mongos`:
   (a) Shut down the `mongos` process.
   (b) Restart it with the 1.8.x binary from the MongoDB Download Page[68].
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[69].
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.

[65] http://downloads.mongodb.org/
[66] http://downloads.mongodb.org/
[67] http://downloads.mongodb.org/
[68] http://downloads.mongodb.org/
[69] http://downloads.mongodb.org/

Journaling Returning to 1.6 after using 1.8 *Journaling* (page 100) 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* (page 100) 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 190) 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 200) enable MongoDB to answer queries entirely from the index when the query only selects fields that the index contains.

Incremental MapReduce Support

The `mapReduce` command supports new options that enable incrementally updating existing *collections*. Previously, a MapReduce job could output either to a temporary collection or to a named permanent collection, which it would overwrite with new data.

You now have several options for the output of your MapReduce jobs:

- You can merge MapReduce output into an existing collection. Output from the Reduce phase will replace existing keys in the output collection if it already exists. Other keys will remain in the collection.
- You can now re-reduce your output with the contents of an existing collection. Each key output by the reduce phase will be reduced with the existing document in the output collection.
- You can replace the existing output collection with the new results of the MapReduce job (equivalent to setting a permanent output collection in previous releases)
- You can compute MapReduce inline and return results to the caller without persisting the results of the job. This is similar to the temporary collections generated in previous releases, except results are limited to 8MB.

For more information, see the `out` field options in the `mapReduce` document.

Additional Changes and Enhancements

1.8.1

- Sharding migrate fix when moving larger chunks.
- Durability fix with background indexing.

- Fixed mongos concurrency issue with many incoming connections.

1.8.0

- All changes from 1.7.x series.

1.7.6

- Bug fixes.

1.7.5

- *Journaling* (page 100).
- 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 190).

1.7.3

- Initial *covered index* (page 200) 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[70], 1.8.0[71]
- 1.7.6[72], 1.7.5[73], 1.7.4[74], 1.7.3[75], 1.7.2[76], 1.7.1[77], 1.7.0[78]

Resources

- MongoDB Downloads[79]
- All JIRA Issues resolved in 1.8[80]

10.2.3 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.

[70] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/v09MbhEm62Y
[71] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/JeHQOnam6Qk
[72] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/3t6GNZ1qGYc
[73] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/S5R0Tx9wkEg
[74] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/9Om3Vuw-y9c
[75] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/DfNUrdbmflI
[76] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/df7mwK6Xixo
[77] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/HUR9zYtTpA8
[78] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/TUnJCg9161A
[79] http://mongodb.org/downloads
[80] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10172

Sharding

Sharding (page 295) 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 295)
- *Deploy a Sharded Cluster* (page 312)
- *Convert a Replica Set to a Replicated Sharded Cluster* (page 341)

Replica Sets

Replica sets (page 217), 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 217)
- *Deploy a Replica Set* (page 259)
- *Convert a Standalone to a Replica Set* (page 262)

Other Improvements

- The `w` option (and `wtimeout`) forces writes to be propagated to `n` servers before returning success (this works especially well with replica sets)
- `$or queries`
- Improved concurrency
- `$slice` operator for returning subsets of arrays
- 64 indexes per collection (formerly 40 indexes per collection)
- 64-bit integers can now be represented in the shell using NumberLong
- The `findAndModify` command now supports upserts. It also allows you to specify fields to return
- $showDiskLoc option to see disk location of a document
- Support for IPv6 and UNIX domain sockets

Installation

- Windows service improvements
- The C++ client is a separate tarball from the binaries

1.6.x Release Notes

- 1.6.5[81]

[81] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/06_QCC05Fpk

1.5.x Release Notes

- 1.5.8[82]
- 1.5.7[83]
- 1.5.6[84]
- 1.5.5[85]
- 1.5.4[86]
- 1.5.3[87]
- 1.5.2[88]
- 1.5.1[89]
- 1.5.0[90]

You can see a full list of all changes on JIRA[91].

Thank you everyone for your support and suggestions!

10.2.4 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* (page 418), in particular the instructions for upgrading the DB format.)

Release 1.4 includes the following improvements over release 1.2:

Core Server Enhancements

- *concurrency* (page 370) improvements
- indexing memory improvements
- *background index creation* (page 191)
- 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`)

[82] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/uJfF1QN6Thk
[83] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/OYvz40RWs90
[84] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/4l0N2U_H0cQ
[85] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/oO749nvTARY
[86] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/380V_Ec_q1c
[87] https://groups.google.com/forum/?hl=en&fromgroups=#!topic/mongodb-user/hsUQL9CxTQw
[88] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/94EE3HVidAA
[89] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/7SBPQ2RSfdM
[90] https://groups.google.com/forum/?fromgroups=#!topic/mongodb-user/VAhJcjDGTy0
[91] https://jira.mongodb.org/secure/IssueNavigator.jspa?mode=hide&requestId=10107

- replication handles clock skew on master
- `$inc` replication fixes
- sharding alpha 3 - notably 2-phase commit on config servers

Deployment and Production

- *configure "slow threshold" for profiling*
- 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 233) 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 209)
- geo `$center` and `$box` searches

10.2.5 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

- `mongoimportjson` has been removed and is replaced with `mongoimport` that can do json/csv/tsv

field filter changing

- We've changed the semantics of the field filter a little bit. Previously only objects with those fields would be returned. Now the field filter only changes the output, not which objects are returned. If you need that behavior, you can use `$exists`

10.3 Other MongoDB Release Notes

10.3.1 Default Write Concern Change

These release notes outline a change to all driver interfaces released in November 2012. See release notes for specific drivers for additional information.

Changes

As of the releases listed below, there are two major changes to all drivers:

1. All drivers will add a new top-level connection class that will increase consistency for all MongoDB client interfaces.

   This change is non-backward breaking: existing connection classes will remain in all drivers for a time, and will continue to operate as expected. However, those previous connection classes are now deprecated as of these releases, and will eventually be removed from the driver interfaces.

   The new top-level connection class is named `MongoClient`, or similar depending on how host languages handle namespacing.

2. The default write concern on the new `MongoClient` class will be to acknowledge all write operations [92]. This will allow your application to receive acknowledgment of all write operations.

   See the documentation of *Write Concern* (page 38) 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 419). 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

10.4 Version Numbers

There are three numbers in a MongoDB version:

```
<major version number>.<release number>.<revision number>
```

The second number, the release number, indicates release stability. An odd release number indicates a release in a development series. Even numbered releases indicate a stable, general availability, release.

Additionally, the three numbers indicate the following:

- `<major version number>`: This rarely changes. A change to this number indicates very large changes to MongoDB.
- `<release number>`: A release can include many changes, including new features and updates. Some changes may break backwards compatibility. See release notes for every version for compatibility notes. Even-numbered release numbers are stable branches. Odd-numbered release numbers are development branches.
- `<revision number>`: This digit increments for every release. Changes each revision address bugs and security issues,

[92] 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 `http://docs.mongodb.org/manual/reference/replica-configuration`.

Example

Version numbers:

- 1.0.0 : First stable release
- 1.0.x : Bug fixes to 1.0.x. These releases carry low risk. Always upgrade to the latest revision in your release series.
- 1.1.x : Development release. Includes new features not fully finished and other works-in-progress. Some things may be different than 1.0
- 1.2.x : Second stable release. This is a culmination of the 1.1.x development series.

CHAPTER 11

About MongoDB Documentation

The MongoDB Manual[1] contains comprehensive documentation on the MongoDB *document*-oriented database management system. This page describes the manual's licensing, editions, and versions, and describes how to make a change request and how to contribute to the manual.

For more information on MongoDB, see MongoDB: A Document Oriented Database[2]. To download MongoDB, see the downloads page[3].

11.1 License

This manual is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported[4]" (i.e. "CC-BY-NC-SA") license.

11.2 Editions

In addition to the MongoDB Manual[5], you can also access this content in the following editions:

- ePub Format[6]
- Single HTML Page[7]
- PDF Format[8]

You also can access PDF files that contain subsets of the MongoDB Manual:

- MongoDB Reference Manual[9]
- MongoDB CRUD Operation Introduction[10]

MongoDB Reference documentation is also available as part of dash[11].

[1] http://docs.mongodb.org/manual/#
[2] http://www.mongodb.org/about/
[3] http://www.mongodb.org/downloads
[4] http://creativecommons.org/licenses/by-nc-sa/3.0/
[5] http://docs.mongodb.org/manual/#
[6] http://docs.mongodb.org/v2.2/MongoDB-Manual.epub
[7] http://docs.mongodb.org/v2.2/single/
[8] http://docs.mongodb.org/v2.2/MongoDB-Manual.pdf
[9] http://docs.mongodb.org/v2.2/MongoDB-reference-manual.pdf
[10] http://docs.mongodb.org/v2.2/MongoDB-crud-guide.pdf
[11] http://kapeli.com/dash

11.3 Version and Revisions

This version of the manual reflects version 2.2 of MongoDB.

See the MongoDB Documentation Project Page[12] 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[13].

This edition reflects "`v2.2`" branch of the documentation as of the "**|commit|**" revision. This branch is explicitly accessible via "http://docs.mongodb.org/v2.2" and you can always reference the commit of the current manual in the release.txt[14] file.

The most up-to-date, current, and stable version of the manual is always available at "http://docs.mongodb.org/manual/".

11.4 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[15].

11.5 Contribute to the Documentation

11.5.1 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 Agreement[16], and
- join the mongodb-translators[17] user group.

The mongodb-translators[18] 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 424)
- *Style Guide and Documentation Conventions* (page 425)
- *MongoDB Manual Organization* (page 434)
- *MongoDB Documentation Practices and Processes* (page 431)

[12] http://docs.mongodb.org
[13] https://github.com/mongodb/docs
[14] http://docs.mongodb.org/v2.2/release.txt
[15] https://jira.mongodb.org/browse/DOCS
[16] http://www.mongodb.com/legal/contributor-agreement
[17] http://groups.google.com/group/mongodb-translators
[18] http://groups.google.com/group/mongodb-translators

- *MongoDB Documentation Build System* (page 436)

The entire documentation source for this manual is available in the mongodb/docs repository[19], which is one of the MongoDB project repositories on GitHub[20].

To contribute to the documentation, you can open a GitHub account[21], fork the mongodb/docs repository[22], make a change, and issue a pull request.

In order for the documentation team to accept your change, you must complete the MongoDB Contributor Agreement[23].

You can clone the repository by issuing the following command at your system shell:

```
git clone git://github.com/mongodb/docs.git
```

11.5.2 About the Documentation Process

The MongoDB Manual uses Sphinx[24], a sophisticated documentation engine built upon Python Docutils[25]. The original reStructured Text[26] files, as well as all necessary Sphinx extensions and build tools, are available in the same repository as the documentation.

For more information on the MongoDB documentation process, see:

Style Guide and Documentation Conventions

This document provides an overview of the style for the MongoDB documentation stored in this repository. The overarching goal of this style guide is to provide an accessible base style to ensure that our documentation is easy to read, simple to use, and straightforward to maintain.

For information regarding the MongoDB Manual organization, see *MongoDB Manual Organization* (page 434).

Document History

2011-09-27: Document created with a (very) rough list of style guidelines, conventions, and questions.

2012-01-12: Document revised based on slight shifts in practice, and as part of an effort of making it easier for people outside of the documentation team to contribute to documentation.

2012-03-21: Merged in content from the Jargon, and cleaned up style in light of recent experiences.

2012-08-10: Addition to the "Referencing" section.

2013-02-07: Migrated this document to the manual. Added "map-reduce" terminology convention. Other edits.

2013-11-15: Added new table of preferred terms.

[19] https://github.com/mongodb/docs
[20] http://github.com/mongodb
[21] https://github.com/
[22] https://github.com/mongodb/docs
[23] http://www.mongodb.com/contributor
[24] http://sphinx-doc.org//
[25] http://docutils.sourceforge.net/
[26] http://docutils.sourceforge.net/rst.html

Naming Conventions

This section contains guidelines on naming files, sections, documents and other document elements.

- File naming Convention:
  - For Sphinx, all files should have a `.txt` extension.
  - Separate words in file names with hyphens (i.e. `-`.)
  - For most documents, file names should have a terse one or two word name that describes the material covered in the document. Allow the path of the file within the document tree to add some of the required context/categorization. For example it's acceptable to have `http://docs.mongodb.org/manual/core/sharding.rst` and `http://docs.mongodb.org/manual/administration/sharding.rst`.
  - For tutorials, the full title of the document should be in the file name. For example, `http://docs.mongodb.org/manual/tutorial/replace-one-configuration-server-in-a-sharc`
- Phrase headlines and titles so that they the content contained within the section so that users can determine what questions the text will answer, and material that it will address without needing them to read the content. This shortens the amount of time that people spend looking for answers, and improvise search/scanning, and possibly "SEO."
- Prefer titles and headers in the form of "Using foo" over "How to Foo."
- When using target references (i.e. `:ref:` references in documents,) use names that include enough context to be intelligible thought all documentations. For example, use "`replica-set-secondary-only-node`" as opposed to "`secondary-only-node`". This is to make the source more usable and easier to maintain.

Style Guide

This includes the local typesetting, English, grammatical, conventions and preferences that all documents in the manual should use. The goal here is to choose good standards, that are clear, and have a stylistic minimalism that does not interfere with or distract from the content. A uniform style will improve user experience, and minimize the effect of a multi-authored document.

Punctuation

- Use the oxford comma.

  Oxford commas are the commas in a list of things (e.g. "something, something else, and another thing") before the conjunction (e.g. "and" or "or.").
- Do not add two spaces after terminal punctuation, such as periods.
- Place commas and periods inside quotation marks.
- Use title case for headings and document titles. Title case capitalizes the first letter of the first, last, and all significant words.

Verbs Verb tense and mood preferences, with examples:

- **Avoid** the first person. For example do not say, "We will begin the backup process by locking the database," or "I begin the backup process by locking my database instance,"
- **Use** the second person. "If you need to back up your database, start by locking the database first." In practice, however, it's more concise to imply second person using the imperative, as in "Before initiating a backup, lock the database."

- When indicated, use the imperative mood. For example: "Backup your databases often" and "To prevent data loss, back up your databases."
- The future perfect is also useful in some cases. For example, "Creating disk snapshots without locking the database will lead to an inconsistent state."
- Avoid helper verbs, as possible, to increase clarity and concision. For example, attempt to avoid "this does foo" and "this will do foo" when possible. Use "does foo" over "will do foo" in situations where "this foos" is unacceptable.

Referencing

- To refer to future or planned functionality in MongoDB or a driver, *always* link to the Jira case. The Manual's `conf.py` provides an `:issue:` role that links directly to a Jira case (e.g. `:issue:\`SERVER-9001\``).
- For non-object references (i.e. functions, operators, methods, database commands, settings) always reference only the first occurrence of the reference in a section. You should *always* reference objects, except in section headings.
- Structure references with the *why* first; the link second.

  For example, instead of this:

  Use the *Convert a Replica Set to a Replicated Sharded Cluster* (page 341) 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 341).

General Formulations

- Contractions are acceptable insofar as they are necessary to increase readability and flow. Avoid otherwise.
- Make lists grammatically correct.
  - Do not use a period after every item unless the list item completes the unfinished sentence before the list.
  - Use appropriate commas and conjunctions in the list items.
  - Typically begin a bulleted list with an introductory sentence or clause, with a colon or comma.
- The following terms are one word:
  - standalone
  - workflow
- Use "unavailable," "offline," or "unreachable" to refer to a `mongod` instance that cannot be accessed. Do not use the colloquialism "down."
- Always write out units (e.g. "megabytes") rather than using abbreviations (e.g. "MB".)

Structural Formulations

- There should be at least two headings at every nesting level. Within an "h2" block, there should be either: no "h3" blocks, 2 "h3" blocks, or more than 2 "h3" blocks.
- Section headers are in title case (capitalize first, last, and all important words) and should effectively describe the contents of the section. In a single document you should strive to have section titles that are not redundant and grammatically consistent with each other.

- Use paragraphs and paragraph breaks to increase clarity and flow. Avoid burying critical information in the middle of long paragraphs. Err on the side of shorter paragraphs.
- Prefer shorter sentences to longer sentences. Use complex formations only as a last resort, if at all (e.g. compound complex structures that require semi-colons).
- Avoid paragraphs that consist of single sentences as they often represent a sentence that has unintentionally become too complex or incomplete. However, sometimes such paragraphs are useful for emphasis, summary, or introductions.

  As a corollary, most sections should have multiple paragraphs.
- For longer lists and more complex lists, use bulleted items rather than integrating them inline into a sentence.
- Do not expect that the content of any example (inline or blocked,) will be self explanatory. Even when it feels redundant, make sure that the function and use of every example is clearly described.

ReStructured Text and Typesetting

- Place spaces between nested parentheticals and elements in JavaScript examples. For example, prefer `{ [ a, a, a ] }` over `{[a,a,a]}`.
- For underlines associated with headers in RST, use:
  - = for heading level 1 or h1s. Use underlines and overlines for document titles.
  - − for heading level 2 or h2s.
  - ~ for heading level 3 or h3s.
  - ` for heading level 4 or h4s.
- Use hyphens (-) to indicate items of an ordered list.
- Place footnotes and other references, if you use them, at the end of a section rather than the end of a file.

  Use the footnote format that includes automatic numbering and a target name for ease of use. For instance a footnote tag may look like: `[#note]_` with the corresponding directive holding the body of the footnote that resembles the following: `.. [#note]`.

  Do **not** include `.. code-block:: [language]` in footnotes.
- As it makes sense, use the `.. code-block:: [language]` form to insert literal blocks into the text. While the double colon, `::`, is functional, the `.. code-block:: [language]` form makes the source easier to read and understand.
- For all mentions of referenced types (i.e. commands, operators, expressions, functions, statuses, etc.) use the reference types to ensure uniform formatting and cross-referencing.

Jargon and Common Terms

Preferred Term	Concept	Dispreferred Alternatives	Notes
document	A single, top-level object/record in a MongoDB collection.	record, object, row	Prefer document over object because of concerns about cross-driver langauge handling of objects. Reserve record for "allocation" of storage. Avoid "row," as possible.
database	A group of collections. Refers to a group of data files. This is the "logical" sense of the term "database."		Avoid genericizing "database." Avoid using database to refer to a server process or a data set. This applies both to the datastoring contexts as well as other (related) operational contexts (command context, authentication/authorization context.)
instance	A daemon process. (e.g. **mongos** or **mongod**)	process (acceptable sometimes), node (never acceptable), server.	Avoid using instance, unless it modifies something specifically. Having a descriptor for a process/instance makes it possible to avoid needing to make mongod or mongos plural. Server and node are both vague and contextually difficult to disambiguate with regards to application servers, and underlying hardware.
field name	The identifier of a value in a document.	key, column	Avoid introducing unrelated terms for a single field. In the documentation we've rarely had to discuss the identifier of a field, so the extra word here isn't burdensome.
field/value	The name/value pair that describes a unit of data in MongoDB.	key, slot, attribute	Use to emphasize the difference between the name of a field and its value For exdample, "_id" is the field and the default value is an ObjectId.
value	The data content of a field.	data	
MongoDB	A group of processes, or deployment that implement the MongoDB interface.	mongo, mongodb, cluster	Stylistic preference, mostly. In some cases it's useful to be able to refer generically to instances (that may be either **mongod** or **mongos**.)
subdocument	An embeded or nested document within a document or an array.	embeded document, nested document	
mapreduce	An operation performed by the mapReduce command.	mapReduce, map reduce, map/reduce	Avoid confusion with the command, shell helper, and driver interfaces. Makes it possible to discuss the operation generally.
cluster	A sharded cluster.	grid, shard cluster, set, deployment	Cluster is a great word for a group of processes; however, it's important to avoid letting the term become generic. Do not use for any group of MongoDB processes or deployments.
sharded cluster	A *sharded cluster*.	shard cluster, cluster, sharded system	
replica set	A deployment of replicating **mongod** programs that provide redundancy and automatic failover.	set, replication deployment	
deployment	A group of MongoDB processes, or a standalone **mongod** instance.	cluster, system	Typically in the form MongoDB deployment. Includes standalones, replica sets and sharded clusters.
data set	The collection of phyisical databases provided by a MongoDB deployment.	database, data	Important to keep the distinction between the data provided by a mongod or a sharded cluster as distinct from each "database" (i.e. a logical

Database Systems and Processes

- To indicate the entire database system, use "MongoDB," not mongo or Mongo.
- To indicate the database process or a server instance, use `mongod` or `mongos`. Refer to these as "processes" or "instances." Reserve "database" for referring to a database structure, i.e., the structure that holds collections and refers to a group of files on disk.

Distributed System Terms

- Refer to partitioned systems as "sharded clusters." Do not use shard clusters or sharded systems.
- Refer to configurations that run with replication as "replica sets" (or "master/slave deployments") rather than "clusters" or other variants.

Data Structure Terms

- "document" refers to "rows" or "records" in a MongoDB database. Potential confusion with "JSON Documents."

  Do not refer to documents as "objects," because drivers (and MongoDB) do not preserve the order of fields when fetching data. If the order of objects matter, use an array.
- "field" refers to a "key" or "identifier" of data within a MongoDB document.
- "value" refers to the contents of a "field".
- "sub-document" describes a nested document.

Other Terms

- Use `example.net` (and `.org` or `.com` if needed) for all examples and samples.
- Hyphenate "map-reduce" in order to avoid ambiguous reference to the command name. Do not camel-case.

Notes on Specific Features

- Geo-Location
  1. While MongoDB *is capable* of storing coordinates in sub-documents, in practice, users should only store coordinates in arrays. (See: DOCS-41[27].)

MongoDB Documentation Practices and Processes

This document provides an overview of the practices and processes.

Commits

When relevant, include a Jira case identifier in a commit message. Reference documentation cases when applicable, but feel free to reference other cases from jira.mongodb.org[28].

Err on the side of creating a larger number of discrete commits rather than bundling large set of changes into one commit.

[27] https://jira.mongodb.org/browse/DOCS-41
[28] http://jira.mongodb.org/

For the sake of consistency, remove trailing whitespaces in the source file.

"Hard wrap" files to between 72 and 80 characters per-line.

Standards and Practices

- At least two people should vet all non-trivial changes to the documentation before publication. One of the reviewers should have significant technical experience with the material covered in the documentation.
- All development and editorial work should transpire on GitHub branches or forks that editors can then merge into the publication branches.

Collaboration

To propose a change to the documentation, do either of the following:

- Open a ticket in the documentation project[29] 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[30], fork the mongodb/docs repository[31], 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[32] and docutils can catch numerous errors in the format and syntax of the documentation. Additionally, having access to an example documentation as it *will* appear to the users is useful for providing more effective basis for the review process. Besides Sphinx, Pygments, and Python-Docutils, the documentation repository contains all requirements for building the documentation resource.

Talk to someone on the documentation team if you are having problems running builds yourself.

Publication

The makefile for this repository contains targets that automate the publication process. Use `make html` to publish a test build of the documentation in the `build/` directory of your repository. Use `make publish` to build the full contents of the manual from the current branch in the `../public-docs/` directory relative the docs repository.

Other targets include:

- `man` - builds UNIX Manual pages for all Mongodb utilities.
- `push` - builds and deploys the contents of the `../public-docs/`.
- `pdfs` - builds a PDF version of the manual (requires LaTeX dependencies.)

Branches

This section provides an overview of the git branches in the MongoDB documentation repository and their use.

[29] https://jira.mongodb.org/browse/DOCS
[30] https://github.com/
[31] https://github.com/mongodb/docs
[32] 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[33] 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[34] or contact the documentation team[35].

[33] http://jira.mongodb.org/browse/DOCS
[34] https://jira.mongodb.org/browse/DOCS
[35] 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.

Additionally, in the top level of the `source/` directory, there are a number of "topical" index or outline files. These (like the "index" and "contents" files) use the `.. toctree::` directive to provide organization within the documentation. The topical indexes combine to create the index in the contents file.

Topical Indexes and Meta Organization Because the documentation on any given subject exists in a number of different locations across the resource the "topical" indexes provide the real structure and organization to the resource. This organization makes it possible to provide great flexibility while still maintaining a reasonable organization of files and URLs for the documentation. Consider the following example:

> Given that topic such as "replication," has material regarding the administration of replica sets, as well as reference material, an overview of the functionality, and operational tutorials, it makes more sense to include a few locations for documents, and use the meta documents to provide the topic-level organization.

Current topical indexes include:

- getting-started
- administration
- applications
- reference
- mongo
- sharding
- replication
- faq

Additional topical indexes are forthcoming.

The Top Level Folders

The documentation has a number of top-level folders, that hold all of the content of the resource. Consider the following list and explanations below:

- "administration" - contains all of the operational and architectural information that systems and database administrators need to know in order to run MongoDB. Topics include: monitoring, replica sets, shard clusters, deployment architectures, and configuration.
- "applications" - contains information about application development and use. While most documentation regarding application development is within the purview of the driver documentation, there are some larger topics regarding the use of these features that deserve some coverage in this context. Topics include: drivers, schema design, optimization, replication, and sharding.

- "core" - contains overviews and introduction to the core features, functionality, and concepts of MongoDB. Topics include: replication, sharding, capped collections, journaling/durability, aggregation.
- "reference" - contains references and indexes of shell functions, database commands, status outputs, as well as manual pages for all of the programs come with MongoDB (e.g. `mongostat` and `mongodump`.)
- "tutorial" - contains operational guides and tutorials that lead users through common tasks (administrative and conceptual) with MongoDB. This includes programming patterns and operational guides.
- "faq" - contains all the frequently asked questions related to MongoDB, in a collection of topical files.

MongoDB Documentation Build System

This document contains more direct instructions for building the MongoDB documentation.

Getting Started

Install Dependencies The MongoDB Documentation project depends on the following tools:

- Python
- Git
- Inkscape (Image generation.)
- LaTeX/PDF LaTeX (typically texlive; for building PDFs)
- Giza[36]

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[37].

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[38]. 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:

[36] https://pypi.python.org/pypi/giza
[37] http://inkscape.org/download/
[38] http://www.tug.org/mactex/2011/

```
pacman -S texlive-bin texlive-core texlive-latexextra
```

Debian/Ubuntu Install the required system packages with the following command:

```
apt-get install inkscape python-pip
```

Then install the following Python packages:

```
pip install giza
```

Optional

To generate PDFs for the full production build, install the following packages from the system repository:

```
apt-get install texlive-latex-recommended texlive-latex-recommended
```

Setup and Configuration Clone the repository:

```
git clone git://github.com/mongodb/docs.git
```

Building the Documentation

The MongoDB documentation build system is entirely accessible via `make` targets. For example, to build an HTML version of the documentation issue the following command:

```
make html
```

You can find the build output in `build/<branch>/html`, where `<branch>` is the name of the current branch.

In addition to the `html` target, the build system provides the following targets:

`publish` Builds and integrates all output for the production build. Build output is in `build/public/<branch>/`. When you run `publish` in the `master`, the build will generate some output in `build/public/`.

`push; stage` Uploads the production build to the production or staging web servers. Depends on `publish`. Requires access production or staging environment.

`push-all; stage-all` Uploads the entire content of `build/public/` to the web servers. Depends on `publish`. Not used in common practice.

`push-with-delete; stage-with-delete` Modifies the action of `push` and `stage` to remove remote file that don't exist in the local build. Use with caution.

`html; latex; dirhtml; epub; texinfo; man; json` These are standard targets derived from the default Sphinx Makefile, with adjusted dependencies. Additionally, for all of these targets you can append `-nitpick` to increase Sphinx's verbosity, or `-clean` to remove all Sphinx build artifacts.

`latex` performs several additional post-processing steps on `.tex` output generated by Sphinx. This target will also compile PDFs using `pdflatex`.

`html` and `man` also generates a `.tar.gz` file of the build outputs for inclusion in the final releases.

If you have any questions, please feel free to open a Jira Case[39].

[39] https://jira.mongodb.org/browse/DOCS